Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica

Sperimentazione e valutazione di OpenCL su piattaforme parallele per un metodo a volumi finiti per le acque basse

Relatore: prof. Luca Oddone Breveglieri

Correlatore: prof. Edie Miglio

Laureando: Roberto Stramare

Matricola: 718754

Anno accademico: 2010/2011

Appello di aprile 2012

Ringraziamenti

Si ringraziano il prof. Edie Miglio per aver proposto questo argomento e avermi permesso di sviluppare la tesi presso il Dipartimento di Matematica, il prof. Luca Breveglieri per aver accettato la proposta di tesi, Mattia Penati e Alessandro Barenghi per i consigli sull'implementazione, il Dipartimento di Matematica, i Servizi per il Diritto allo Studio e infine la mia famiglia per il sostegno. Se ho dimenticato qualcuno ringrazio anche lui. Si ringraziano anche tutti coloro che in qualunque modo mi hanno aiutato a concludere gli studi precedenti alla Laurea Specialistica, senza di loro non sarei arrivato a raggiungere questo risultato.

Indice

1 Modellistica **14**
- 1.1 Introduzione ai modelli dei fluidi . . . 14
- 1.2 Tipi di onde e classificazione dei modelli . . . 15
- 1.3 Equazioni di Navier-Stokes . . . 15
- 1.4 Shallow water equations . . . 19
  - 1.4.1 Applicazioni . . . 22

2 Metodi di risoluzione **23**
- 2.1 Formulazione generale delle equazioni di conservazione . . . 23
- 2.2 La condizione CFL . . . 26
- 2.3 Un flusso instabile . . . 27
- 2.4 Il metodo Lax-Friedrichs . . . 27
- 2.5 Il metodo Lax-Friedrichs per le acque basse . . . 28
- 2.6 Codice Fortran e C . . . 28

3 Calcolo parallelo, GPU computing e OpenCL **34**
- 3.1 Introduzione al calcolo eterogeneo . . . 34
- 3.2 Storia del calcolo parallelo con CPU . . . 35
- 3.3 GPU computing . . . 37
  - 3.3.1 Breve storia delle GPU . . . 37
  - 3.3.2 GPU computing primordiale . . . 38
  - 3.3.3 CUDA . . . 40
    - 3.3.3.1 Cos'è CUDA . . . 40
    - 3.3.3.2 Utilizzo di CUDA . . . 40
    - 3.3.3.3 Applicazioni . . . 42
  - 3.3.4 ATI . . . 43
    - 3.3.4.1 ATI Stream . . . 43
  - 3.3.5 Presente e futuro del calcolo eterogeneo . . . 45
- 3.4 OpenCL . . . 47
  - 3.4.1 Altre aziende che supportano OpenCL . . . 53
- 3.5 Tipi di parallelismo gestibili in OpenCL . . . 53
- 3.6 Differenze essenziali tra CPU e GPU . . . 56
  - 3.6.1 Architettura Cypress delle GPU AMD . . . 58
  - 3.6.2 Architettura CUDA delle GPU Nvidia Fermi . . . 60

4 Implementazioni con OpenCL **63**
4.1 Differenze generali 63
4.2 Codice host 64
4.3 Parallelizzazione 72
4.4 Caratteristiche dipendenti dalla piattaforma 75
4.5 Modifiche introdotte 75
4.5.1 Copia parallela di righe e colonne 75
4.5.2 Uso della memoria locale 77
4.5.3 Riduzione dei trasferimenti di memoria 85
4.5.4 Unione dei metodi precedenti 85
4.5.5 Tecniche Single Instruction Multiple Data (SIMD) 86
4.6 Altre considerazioni 88
4.6.1 Riduzione nel numero di work-item 92
4.7 Implementazione Intel 92
4.8 Considerazioni sull'aggiunta di grafica 93

5 Risultati dei test **97**
5.1 Piattaforme di esecuzione e caso di test 97
5.2 Versioni in virgola mobile a doppia precisione 99
5.2.1 Versione C sequenziale con vettori di vettori per rappresentare matrici 99
5.2.2 Versione C sequenziale con vettori per rappresentare matrici 100
5.2.3 Prima versione C con OpenCL 101
5.2.4 Versione con copia parallela di righe e colonne 102
5.2.5 Versione con uso di memoria locale e copie successive da memoria globale 102
5.2.6 Versione con uso di memoria locale 103
5.2.7 Versione con uso di memoria locale e copia parallela di righe e colonne 104
5.2.8 Versione con uso di memoria locale senza copia dei buffer 104
5.2.9 Versione con uso di memoria locale senza copia dei buffer con copia parallela di righe e colonne 105
5.2.10 Versione con numero ridotto di work-item 107
5.2.11 Scalabilità per dimensione della griglia a discretizzazione variabile 107
5.3 Versioni a virgola mobile in singola precisione 110
5.3.1 Versione C sequenziale con vettori di vettori per rappresentare matrici 110
5.3.2 Versione C sequenziale con vettori per rappresentare matrici 110
5.3.3 Prima versione C con OpenCL 111
5.3.4 Versione con copia parallela di righe e colonne 111
5.3.4.1 Versione con copia parallela e float4 di righe e copia parallela di colonne 112
5.3.5 Versione con uso di memoria locale e copie successive da memoria globale 112
5.3.6 Versione con uso di memoria locale 113

5.3.7 Versione con uso di memoria locale e copia parallela di righe e colonne . . . 113
5.3.8 Versione con uso di memoria locale senza copia dei buffer 113
5.3.9 Versione con uso di memoria locale senza copia dei buffer e copia parallela di righe e colonne . . . 114
5.3.10 Versione con numero ridotto di work-item . . . 116
5.3.11 Scalabilità per dimensione della griglia a discretizzazione variabile . . . 116
5.4 Riassunto . . . 119

6 Conclusioni **120**
6.1 Considerazioni finali e confronto con altre implementazioni . . . 120
6.2 Sviluppi futuri . . . 121

Elenco delle figure

1.1 Superfici di riferimento per la descrizione di un fenomeno fluidodinamico 19

2.1 Esempio di output Paraview per una simulazione in doppia precisione di dimensione 100x100 e discretizzazione 5 al passo 100 (t=5.0507627227611≈5.051) di 396. Nella legenda dell'immagine la virgola è usata come separatore della parte frazionaria. 33

3.1 Il processore Intel a 80 core 37
3.2 Input e output dell'applicazione GPUoceto per la conversione di kernel PTX 41
3.3 La GPU Tesla C1060 42
3.4 ATI RV770 corrispondente alle GPU Radeon HD: 4870 (RV770XT), 4850 (RV770PRO) e 4830 (RV770PROLE) 44
3.5 La catena di compilazione OpenCL nell'AMD APP SDK (e nel precedente ATI Stream SDK) 45
3.6 Il microprocessore IBM Cell Broadband Engine 46
3.7 Struttura della memoria secondo lo standard OpenCL 50
3.8 Interazioni tra i diversi tipi di memoria nell'architettura ATI Stream 52
3.9 Flusso standard tra host e GPU dei dati in OpenCL 52
3.10 Architettura convenzionale di una CPU 56
3.11 Architettura di una moderna GPU 57
3.12 Architettura Cypress, struttura della GPU 59
3.13 Struttura di un AMD *stream core* 59
3.14 Architettura della memoria AMD per più SIMD Engine 60
3.15 Architettura CUDA Fermi, struttura di uno Stream Multiprocessor 61
3.16 Struttura di un core CUDA 61
3.17 Gerarchia di memoria Nvidia per un singolo SM e comunicazione con memoria globale 62

4.1 Esempio di memoria locale 84
4.2 Simulazione Intel Shallow Water originale eseguita su CPU . . . 93
4.3 Simulazione Intel Shallow Water eseguita su GPU 94
4.4 Seconda simulazione Intel Shallow Water eseguita su GPU 95

5.1 Esempio di output Paraview per una simulazione in doppia precisione di dimensione 1000x1000 e discretizzazione 0.5 al passo 1000 (t=5,050762722761≈5,051) di 3960. Nella legenda dell'immagine la virgola è usata come separatore della parte frazionaria. 100
5.2 Grafico della scalabilità a discretizzazione variabile in doppia precisione 109
5.3 Grafico della scalabilità a discretizzazione variabile in singola precisione 118

Elenco delle tabelle

3.1 Tassonomia di Flynn per le architetture parallele 54

5.1 Tempi di esecuzione della versione sequenziale C con puntatori a puntatori in doppia precisione 99
5.2 Tempi di esecuzione della versione sequenziale C con vettori puri in doppia precisione 100
5.3 Tempi di esecuzione della prima parallelizzazione in doppia precisione utilizzante OpenCL 101
5.4 Tempi di esecuzione della versione OpenCL in doppia precisione con copia parallela di righe e colonne 102
5.5 Tempi di esecuzione del primo tentativo di utilizzo di memoria locale in doppia precisione 102
5.6 Tempi di esecuzione del secondo modo di utilizzo di memoria locale in doppia precisione 103
5.7 Tempi di esecuzione della versione in doppia precisione con utilizzo di memoria locale e copia parallela di righe e colonne 104
5.8 Tempi di esecuzione della versione in doppia precisione con utilizzo di memoria locale senza copia dei buffer 104
5.9 Tempi di esecuzione della versione in doppia precisione con utilizzo di memoria locale senza copia dei buffer e copia parallela di righe e colonne 105
5.10 Tempi di esecuzione in doppia precisione per Radeon HD 5830 per work-group (e memoria locale) non quadrati 105
5.11 Tempi di esecuzione in doppia precisione per GTS450 per work-group (e memoria locale) non quadrati e 25x25 106
5.12 Tempi di esecuzione in doppia precisione per Tesla C1060 per work-group non quadrati di dimensione 384, 256 e 32 e dimensioni quadrate 19x19 e 16x16 106
5.13 Tempi di esecuzione della versione utilizzante OpenCL con numero di work-item pari a quello di computation units (o core) . . 107
5.14 Prove di scalabilità a discretizzazione variabile in doppia precisione 108
5.15 Tempi di esecuzione del singolo passo per le prove di scalabilità a discretizzazione variabile in doppia precisione 108

5.16 Tempi di esecuzione della versione sequenziale C con puntatori a puntatori in singola precisione 110
5.17 Tempi di esecuzione della versione sequenziale C con vettori puri in singola precisione 110
5.18 Tempi di esecuzione della prima parallelizzazione in singola precisione utilizzante OpenCL 111
5.19 Tempi di esecuzione della versione OpenCL in singola precisione con copia parallela di righe e colonne 111
5.20 Tempi di esecuzione della versione OpenCL in singola precisione con copia parallela float4 di righe e copia parallela di colonne . . 112
5.21 Tempi di esecuzione del primo tentativo di utilizzo di memoria locale in singola precisione 112
5.22 Tempi di esecuzione del secondo modo di utilizzo di memoria locale in singola precisione 113
5.23 Tempi di esecuzione della versione in singola precisione con utilizzo di memoria locale e copia parallela di righe e colonne 113
5.24 Tempi di esecuzione della versione in singola precisione con utilizzo di memoria locale senza copia dei buffer 113
5.25 Tempi di esecuzione della versione in singola precisione con utilizzo di memoria locale senza copia dei buffer e copia parallela di righe e colonne 114
5.26 Tempi di esecuzione in singola precisione per Radeon HD 5830 per work-group (e memoria locale) non quadrati 114
5.27 Tempi di esecuzione in singola precisione per GTS450 per work-group (e memoria locale) non quadrati di dimensione 1024, 512, 256 e 32 115
5.28 Tempi di esecuzione in singola precisione per Tesla C1060 per work-group non quadrati di dimensione 512, 256 e 32 e dimensione quadrata 16x16 116
5.29 Tempi di esecuzione della versione utilizzante OpenCL con numero di work-item pari a quello di computation units (o core) . . 116
5.30 Prove di scalabilità a discretizzazione variabile in singola precisione117
5.31 Tempi di esecuzione del singolo passo per le prove di scalabilità a discretizzazione variabile in singola precisione 118

Elenco degli algoritmi

2.1 Codice Fortran per il Metodo Lax-Friedrichs per la simulazione delle acque basse 29
2.2 Codice C per il Metodo Lax-Friedrichs per la simulazione delle acque basse 31
4.1 Codice C del kernel OpenCL per l'esecuzione parallela del metodo Lax-Friedrichs per le acque basse 73

Sommario

La simulazione di problemi scientifici e ingegneristici richiede grandi capacità computazionali, questa tesi mostrerà l'uso dello standard OpenCL (Open Computing Language) per il calcolo su *hardware* parallelo di diverso tipo (quindi anche dispositivi come GPU programmabili oltre a CPU *multicore*) in un'implementazione parallela sviluppata a questo scopo per la soluzione del problema di simulazione noto come "equazioni delle acque basse" o, in inglese, "shallow water equations". CPU parallele e GPU programmabili sono entrambe dotate di parallelismo a livello *hardware,* ed entrambe sono sempre più diffuse, e in questa tesi si mostrerà quindi come è stato utilizzato OpenCL per migliorare le prestazioni tramite la parallelizzazione del codice (e l'esecuzione su tale *hardware*) della soluzione dello schema di simulazione bidimensionale delle equazioni delle acque basse (denominate in inglese "2D shallow water equations"), questo in particolar modo su GPU effettuando quindi quello che viene definito GPU computing.

Dopo la spiegazione del modello utilizzato per la simulazione, dell'algoritmo per la sua risoluzione, dell'origine del calcolo parallelo, del significato di GPU computing, della sua storia, dello standard OpenCL e delle differenze tra CPU e GPU si presenteranno diverse versioni equivalenti utilizzanti calcolo parallelo tramite OpenCL con la spiegazione delle peculiarità di ognuna. In seguito saranno presentati i risultati sperimentali dell'esecuzione su diversi dispositivi (sia CPU sia GPU, sia su personal computer sia su server) e infine possibili sviluppi dell'utilizzo di OpenCL in questo problema di simulazione.

Introduzione

Negli ultimi anni la facilità di accedere (anche in ambito domestico) a macchine che permettano un reale calcolo parallelo è aumentata, in particolare ora è facile disporre di macchine con processori multicore e con schede grafiche programmabili, CPU *multicore* sono poi contenute anche in personal computer a basso costo e dispositivi mobili e le GPU non sono più da tempo relegate al solo calcolo dell'output grafico e sono anch'esse presenti su dispositivi mobili. L'uso delle GPU per effettuare calcoli paralleli viene chiamato GPGPU (General Purpose GPU) oppure GPU computing. La sigla GPU sta per Graphics Processing Units ed indica il componente centrale di una scheda video, è da notare che questi componenti eseguono normalmente operazioni in modo pesantemente parallelo, tuttavia limitate alla grafica, anche se esistevano comunque modi di utilizzare questa potenza di calcolo nascosta per altri scopi, uno di questi approcci è stato utilizzato in [Hagen et Al., 2005].

Entrambi i maggiori produttori di GPU (ATI ed NVIDIA) hanno sviluppato soluzioni per il GPU computing (rispettivamente con i linguaggi Brook e C for CUDA) che sono state poi dirette verso la definizione di OpenCL, uno standard (inizialmente proposto da Apple Inc.) per utilizzare dispositivi paralleli (in particolare le GPU ma anche le CPU) utilizzando principalmente il linguaggio di programmazione C in modo indipendente dal produttore. La soluzione precedente di ATI è considerata deprecata mentre NVIDIA supporta ancora C for CUDA.

Si fa notare che l'architettura delle schede video è molto adatta al calcolo parallelo, infatti ATI ed NVIDIA oltre a offrire schede video offrono anche schede con architetture simili alle controparti video utilizzabili esclusivamente per il calcolo parallelo ad alte prestazioni (la serie Tesla di Nvidia e la serie Firestream di AMD).

Lo scopo dell'attività di tesi è stato quello di utilizzare OpenCL per lo svilppo di un algoritmo parallelo per la soluzione del problema di simulazione noto come "equazioni delle acque basse" o, in inglese, "shallow water equations". In questa tesi si mostrerà come si può utilizzare OpenCL per realizzare tramite parallelizzazione un'implementazione più efficiente, in particolar modo su GPU, della soluzione dello schema di simulazione bidimensionale delle equazioni delle acque basse (denominate in inglese "2D shallow water equations").

Un approccio simile era già stato utilizzato in [Hagen et Al., 2005] tuttavia il GPU computing non era ancora supportato (e spinto dalle aziende produttrici di

GPU) come oggi e il programma era basato sull'utilizzo di OpenGL. L'impulso a questi tesi è stato dato dalla volontà di migliorare, grazie alla parallelizzazione offerta da OpenCL, il codice Fortran completamente sequenziale realizzato dal prof. Edie Miglio che implementa uno schema di soluzione per le equazioni delle acque basse.

Questa tesi si apre nel Capitolo 1 con una panoramica sui modelli per fluidi incomprimibili a superficie libera e presenta poi le equazioni di Navier-Stokes e come da esse è ricavato il modello bidimensionale delle acque basse. La tesi poi nel Capitolo 2 descrive prima la forma generale delle equazioni di conservazione e forme generali per la loro soluzione e poi il metodo Lax-Friedrichs e come è stato qui applicato. Nel Capitolo 3 la tesi prosegue con la descrizione del calcolo parallelo, del GPU computing e dello standard OpenCL (con anche la loro storia e citando anche diverse piattaforme hardware utilizzate o utilizzabili per il GPU computing, con o senza OpenCL) ed esempi di applicazione; il capitolo termina spiegando i diversi modelli di calcolo parallelo utilizzabili in OpenCL e mostrando le differenze architetturali tra CPU e GPU (con anche la descrizione di due GPU specfiche). Nel Capitolo 4 si presentano diverse versioni equivalenti utilizzanti calcolo parallelo tramite OpenCL con la spiegazione delle peculiarità di ognuna partendo da un'implementazione basilare e descrivendo poi i miglioramenti apportati, si accennano infine ad ulteriori miglioramenti apportabili alle prestazioni e all'output. Nel Capitolo 5 sono presentati i risultati sperimentali dell'esecuzione su diversi dispositivi (sia CPU sia GPU, sia su personal computer sia su server). Infine nel Capitolo 6 si traggono le conclusioni valutando positivamente l'utilizzo di OpenCL per problemi di questo tipo facendo anche il confronto con la scarsa letteratura esistente e si descrivono inoltre possibili sviluppi effettuabili a partire dalle implementazioni descritte in questa tesi.

Capitolo 1

Modellistica

In questo capitolo si presenteranno una breve introduzione sui diversi tipi di modelli per il comportamento dei fluidi a superficie libera, la loro classificazione, le equazioni di Navier-Stokes per fluidi incomprimibili da cui essi sono ricavati, l'utilizzo di queste sotto opportune ipotesi per la costruzione del modello bidimensionale per le acque basse e infine le possibili applicazioni di quest'ultimo.

1.1 Introduzione ai modelli dei fluidi

Esistono diversi modelli per per lo studio del comportamento dei fluidi a superficie libera e sono ricavati dalle equazioni di Navier-Stokes per fluidi incomprimibili. Qui si procederà a un'ilustrazione generale seguendo l'introduzione di [Fontana, 1998].

Le equazioni delle acque poco profonde (Shallow Water Equations o anche 2D-SWE) sono il modello più semplice che è possibile ricavare dalle equazioni di Navier-Stokes. Esse sono ottenute integrando lungo la verticale le equazioni di conservazione della quantità di moto e della massa. Nelle 2D-SWE si introduce l'ipotesi idrostatica sulla pressione: si assume che le accelerazioni verticali siano piccole, ossia si considerano onde di gravità la cui lunghezza è più grande dell'ampiezza (onde lunghe). Questo è il caso in cui le dimensioni orizzontali del dominio sono più grandi di quelle verticali. Le 2D-SWE sono un modello bidimensionale [Fontana, 1998, Agoshkov, 1994]. L'ipotesi idrostatica consente di ridurre il costo computazionale che richiederebbe la discretizzazione dell'equazione della quantità di moto in direzione verticale, infatti elimina le onde acustiche verticali e contemporaneamente trasforma il campo di pressione tridimensionale in uno bidimensionale.

Un altro modello derivato dalla equazioni di Navier-Stokes e con l'ipotesi idrostatica si ottiene integrando lungo la verticale solo l'equazione di continuità. In pratica il dominio lungo la direzione verticale viene suddiviso in strati, in ciascuno dei quali le variabili fisiche variano secondo una legge preassegnata.

Questo modello viene chiamato quindi modello quasi-3D per le Shallow Water o three-dimensional multi-layers Shallow Water Equations o 3D-ML-SWE in inglese.

Si possono ottenere altri modelli rinunciando all'ipotesi idrostatica, il primo denominato Quasi-3D Quasi Hydrostatic (3D-QH-SWE) e il secondo Equazioni di Boussinesq che modellizzano le onde corte [Fontana, 1998].

1.2 Tipi di onde e classificazione dei modelli

I modelli nella terminologia ingegneristica vengono distinti tra modelli a onde lunghe e onde corte [Fontana, 1998].

In idrodinamica si definiscono onde corte in acque poco profonde (short shallow water waves) quelle onde la cui lughezza è grande se confrontata con la profondità dell'acqua nella quale si propagano e conseguentemente esse sono solitamente analizzate come onde lunghe. Le onde sono però conisderate tanto lunghe quanto la profondità h è più piccola della corrispondente lunghezza d'onda cioè

$$\beta = \frac{h^2}{\lambda^2} << 1$$

dove β è una quantità adimensionale che misura la frequenza di dispersione. La formula precedente indica che la variazione di parametri è trascurabile se confrontata con quella orizzontale. Perciò ogni media operata lungo la verticale non causa rilevanti perdite di informazioni. Infine un'onda di superficie viene classificata come onda piccola ma di ampiezza finita quando il rapporto tra l'altezza dell'onda a e la profondità totale è piccola, ossia

$$\alpha = \frac{a}{H} < 1$$

dove α indica l'ampiezza di dispersione.

Sulla base di tali caratteristiche si possono distinguere tre teorie per lo studio dell'evoluzione delle onde di superficie in un fluido.

La prima di queste è la teoria lineare che si occupa di onde infinitesime di lunghezza arbitraria. La seconda è la teoria delle onde lunghe, essa include onde di ampiezza arbitraria con lunghezza infinita rispetto alla profondità. La terza ed ultima teoria è basata sul bilanciamento tra gli effetti lineari e quelli di dispersione, così le onde posscono propagarsi per lughe distanze senza subire cambiamenti di forma. Le equazioni delle acque basse o poco profnde appartengono alla prima categoria, quelle di Boussinesq alla terza.

1.3 Equazioni di Navier-Stokes

Le equazioni delle acque basse sono una forma particolare delle più complesse equazioni di Navier-Stokes che descrivono il moto dei fluidi in generale attraverso un sistema di equazioni alle derivate parziali.

Le equazioni di Navier-Stokes sono ottenute imponendo la conservazione della massa e della quantità di moto ad un fluido (viene indicato in inglese come momentum o linear momentum). In questa sezione si mostrerà tale procedimento rifacendosi a [Dawson and Mirabito, 2008].

Per prima cosa è necessario imporre la conservazione della massa in un volume di controllo, indicato con Ω e definito da una superficie chiusa $\partial\Omega$, con la seguente equazione che rappresenta l'uguaglianza tra variazione istantanea di massa nel volume di controllo e il flusso netto di massa attraverso la superficie $\partial\Omega$ (i vettori sono indicati in grassetto)

$$\frac{d}{dt}\int_{\Omega}\rho dV = -\oint_{\partial\Omega}(\rho\mathbf{v})\cdot\mathbf{n}dA$$

dove

- ρ è la densità del fluido misurata in $\mathrm{kg/m^3}$
- $\mathbf{v} = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$ è un vettore che rappresenta la velocità del fluido le cui componenti sono misurate in $\mathrm{m/s}$
- $\mathbf{n}$ è il vettore normale cioè uscente in ogni punto dalla superficie $\partial\Omega$ perpendicolare alla superficie stessa

All'equazione precedente è possibile applicare il teorema della divergenza (o di Gauss-Green) ottenendo

$$\frac{d}{dt}\int_{\Omega}\rho dV = -\int_{\Omega}\nabla\cdot(\rho\mathbf{v})\,dV$$

assumendo che ρ sia una funzione infinitamente differenziabile (che ammette cioè derivate parziali qualsiasi e di qualsiasi ordine) è possibile applicare la regola di Leibniz (dopo aver portato tutto al primo membro) ottenendo

$$\int_{\Omega}\left[\frac{\partial\rho}{\partial t}dV + -\nabla\cdot(\rho\mathbf{v})\right]dV = 0$$

poiché Ω è arbitrario è possibile rimuovere il segno di integrale ottenendo l'equazione in forma differenziale di conservazione della massa in un fluido nel caso generale

$$\frac{\partial\rho}{\partial t}dV + -\nabla\cdot(\rho\mathbf{v}) = 0 \tag{1.1}$$

In un modo simile, considerando ancora per prima cosa un volume di controllo, si ricava l' equazione della conservazione della quantità di moto

$$\frac{d}{dt}\int_{\Omega}\rho\mathbf{v}dV = -\int_{\partial\Omega}(\rho\mathbf{v})\otimes\mathbf{v}\cdot\mathbf{n}dA + \int_{\Omega}\rho\mathbf{b}dV + \int_{\partial\Omega}\mathbf{T}\mathbf{n}dA$$

dove $\otimes$ rappresenta il prodotto tensoriale e il primo membro rappresenta la variazione istantanea di quantità di moto nel volume di controllo e gli addendi dell'addizione al secondo membro rappresentano rispettivamente: il flusso di quantità di moto attraverso la superficie $\partial\Omega$ che racchiude il volume di controllo, le forze di corpo che agiscono nel volume di controllo e le forze di contatto esterno che agiscono sulla superficie $\partial\Omega$ e dove:

- $\mathbf{b}$ è la densità di forza di corpo su unità di massa misurata in N/kg
- $\mathbf{T}$ è il tensore degli sforzi di Cauchy misurato in $\mathrm{N/m^2}$. Per maggiori dettagli e una dimostrazione di esistenza si vedano [Oden, 2006] e [Panton, 2006].

Applicando come nel caso precedente il teorema della divergenza dopo aver portato tutto al primo membro si ottiene

$$\frac{d}{dt}\int_\Omega \rho\mathbf{v}dV + \int_\Omega \nabla\cdot(\rho\mathbf{v}\otimes\mathbf{v})\,dV - \int_\Omega \rho\mathbf{b}dV - \int_\Omega \nabla\cdot\mathbf{T}dV = 0$$

assumendo che $\rho\mathbf{v}$ sia una funzione infinitamente differenziabile, la regola di Leibniz può essere applicata ottendendo quindi

$$\int_\Omega \left[\frac{\partial}{\partial t}(\rho\mathbf{v}) + \nabla\cdot(\rho\mathbf{v}\otimes\mathbf{v}) - \rho\mathbf{b} - \nabla\cdot\mathbf{T}\right] dV = 0$$

poiché Ω è arbitrario eliminando l'integrale si ottiene la seguente forma differenziale della conservazione della quantità di moto

$$\frac{\partial}{\partial t}(\rho\mathbf{v}) + \nabla\cdot(\rho\mathbf{v}\otimes\mathbf{v}) - \rho\mathbf{b} - \nabla\cdot\mathbf{T} = 0 \tag{1.2}$$

Combinando le due equazioni in forma differenziale 1.1 e 1.2, derivate dalle leggi di conservazione, si ottiene il seguente sistema:

$$\frac{\partial\rho}{\partial t}dV + -\nabla\cdot(\rho\mathbf{v}) = 0$$

$$\frac{\partial}{\partial t}(\rho\mathbf{v}) + \nabla\cdot(\rho\mathbf{v}\otimes\mathbf{v}) - \rho\mathbf{b} - \nabla\cdot\mathbf{T} = 0$$

Tuttavia queste non sono ancora le equazioni di Navier-Stokes, per ottenerle è necessario fare le seguenti assunzioni

- il fluido è incomprimibile, cioè la densità ρ non dipende dalla pressione (che sarà indicata con p), questo non significa ancora che la densità sia costante perché dipende anche dalla salinità e dalla temperatura dell'acqua
- la salinità e la temperatura dell'acqua sono costanti attraverso tutto il nostro dominio

Dai due punti precedenti si ricava che possiamo assumere ρ costante semplificando così le equazioni nel modo seguente

$$\nabla \cdot v = 0$$

$$\frac{\partial}{\partial t}\rho\mathbf{v} + \nabla \cdot (\rho\mathbf{v}\otimes\mathbf{v}) = \rho\mathbf{b} + \nabla \cdot \mathbf{T}$$

Una delle forze che agiscono sicuramente sul fluido è la gravità perciò

$$\rho\mathbf{b} = \rho\mathbf{g} + \rho\mathbf{b}_{others}$$

dove

- $\mathbf{g}$ è il vettore che rappresenta l'accelerazione di gravità;
- $\mathbf{b}_{others}$ sono le forze di corpo (ad esempio la forza di Coriolis nei sistemi di riferimento rotanti) misurate in N/kg, per ora saranno trascurate.

Per un fluido Newtoniano

$$\mathbf{T} = -p\mathbf{I} + \bar{\mathbf{T}}$$

dove p è la pressione misurata in Pascal (Pa) e $\bar{\mathbf{T}}$ è la matrice dei termini di sforzo.

Ora è possibile ottenere la forma finale delle equazioni di Navier-Stokes a partire dal sistema

$$\nabla \cdot \mathbf{v} = 0 \tag{1.3}$$

$$\frac{\partial}{\partial t}\rho\mathbf{v} + \nabla \cdot (\rho\mathbf{v}\otimes\mathbf{v}) = -\nabla p + \rho\mathbf{g} + \nabla \cdot \bar{\mathbf{T}} \tag{1.4}$$

dove la prima viene detta equazione di continuità e la seconda equazione del momento (quantità di moto). In forma estesa, ricordando che u e v sono le componenti orizzontali di $\mathbf{v}$ e w quella verticale, e indicando con τ gli elementi della matrice $\bar{\mathbf{T}}$, sono:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0$$

$$\frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2)}{\partial x} + \frac{\partial(\rho uv)}{\partial y} + \frac{\partial(\rho uw)}{\partial z} = \frac{\partial(\tau_{xx} - p)}{\partial x} + \frac{\partial(\tau_{xy})}{\partial y} + \frac{\partial(\tau_{xz})}{\partial z}$$

$$\frac{\partial(\rho v)}{\partial t} + \frac{\partial(\rho uv)}{\partial x} + \frac{\partial(\rho v^2)}{\partial y} + \frac{\partial(\rho vw)}{\partial z} = \frac{\partial(\tau_{xy})}{\partial x} + \frac{\partial(\tau_{yy} - p)}{\partial y} + \frac{\partial(\tau_{yz})}{\partial z}$$

$$\frac{\partial(\rho w)}{\partial t} + \frac{\partial(\rho uw)}{\partial x} + \frac{\partial(\rho vw)}{\partial y} + \frac{\partial(\rho w^2)}{\partial z} = -\rho g + \frac{\partial(\tau_{xz})}{\partial x} + \frac{\partial(\tau_{yz})}{\partial y} + \frac{\partial(\tau_{zz-p})}{\partial z}$$

La prima equazione è l'equazione di continuità 1.3, le altre 3 sono le componenti dell'equazione del momento 1.4.

1.4 Shallow water equations

I fenomeni fluidodinamici nei quali le scale orizzontali sono predominanti sulle scale verticali rientrano nell'ambito delle acque poco profonde ([Fontana, 1998, Agoshkov, 1994, Wood, 1993]) come nel caso di onde in prossimità della costa (onde lunghe chilometri in profondità di pochi metri). Anche l'atmosfera può essere descritta in questo modo per scale meteorologiche abbastanza ampie [Fontana, 1998].

A partire dalle equazioni di Navier-Stokes considerando un fluido incomprimibile e una profondità bassa rispetto alle dimensioni orizzontali del dominio si può ricavare un modello semplificato chiamato 2D-Shallow Water Equations.

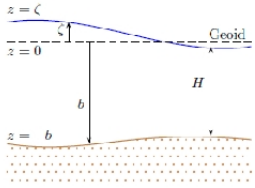
Le equazioni di Navier-Stokes mediate sono definite su un dominio tridimensionale limitato $\widehat{\Omega}$. Nella direzione verticale il fluido è delimitato da due superfici, una superficie libera $z = \eta(x, y, t)$ che rappresenta la superficie dell'acqua che è liquida e cambia con il tempo ed una solida perciò non dipendente dal tempo rappresentante la batimetria del fondale che è caratterizzata dall'equazione $z = -\mathfrak{h}(x, y)$. Entrambe le superfici sono riferite ad un piano di riferimento orizzontale che indichiamo con Ω e poniamo $H = \eta + \mathfrak{h}$ con $H > 0$.

Siccome la superficie libera è un'incognita del problema occorre introdurre un'altra equazione che la descriva: questa equazione esprime la condizione che una particella sulla superficie libera ci rimanga, ossia che il fluido si muova con velocità uguale a quella della superficie stessa (oppure che la velocità sulla superfcie libera non abbia componente normale alla superficie stessa), ed assume la forma seguente:

$$\frac{\partial \eta}{\partial t} + u\frac{\partial \eta}{\partial x} + v\frac{\partial \eta}{\partial y} - w = 0 \tag{1.5}$$

Per ricavare le equazioni è necessario considerare innanzitutto una colonna d'acqua come indicata nella figura seguente tratta da [Dawson and Mirabito, 2008] da cui è anche tratto il resto del procedimento

Figura 1.1: Superfici di riferimento per la descrizione di un fenomeno fluidodinamico



dove

- $\zeta = \zeta(t, x, y)$ è l'elevazione (misurata in metri) relativamente al geoide della superficie libera

- $b = b(x, y)$ è la batimetria (misurata in metri), positivamente verso il basso a partire dal geoide
- $H = H(t, x, y)$ è la profondità totale (misurata in metri) della colonna d'acqua

Si noti che $H = \zeta + b$.

Per descrivere il fluido si impongono per prima cosa le condizioni al contorno:

1. sul fondo $z = -b(x, y)$
   (a) nessun scivolamento cioè $u = v = 0$
   (b) nessun flusso lungo la normale cioè $u\frac{\partial b}{\partial x} + v\frac{\partial b}{\partial y} + w = 0$
   (c) sforzo di taglio sul fondo $\tau_{bx} = \tau_{xx}\frac{\partial b}{\partial x} + \tau_{xy}\frac{\partial b}{\partial y} + \tau_{xy}$ dove τ_{bx} è la frizione sul fondo specificata, si impone una condizione simile per τ_{by}
2. sulla superficie libera $z = \zeta$
   (a) nessun flusso normale cioè $\frac{\partial \zeta}{\partial t} + u\frac{\partial \zeta}{\partial x} + v\frac{\partial \zeta}{\partial y} - w = 0$ che è la 1.5 con una diversa notazione
   (b) $p = 0$ (in [Kubatko, 2005])
   (c) sforzo di taglio sulla superficie (in maniera simile a quanto fatto sul fondo) $\tau_{sx} = -\tau_{xx}\frac{\partial \zeta}{\partial x} + \tau_{xy}\frac{\partial \zeta}{\partial y} + \tau_{xy}$, si impone una condizione simile per τ_{by}

Prima di integrare sulla profondità è necessario esaminare l'equazione 1.4 del momento (quantità di moto) per la componente verticale di velocità. Per ordine di grandezza tutti i termini possono esssere trascurati eccetto la derivata della pressione e la forza di gravità. L'equazione dello z-momento (cioè del momento lungo l'asse z) si riduce quindi a

$$\frac{\partial p}{\partial z} = \rho g$$

dove g è qui uno scalare poiché stiamo considerando solo l'asse z. Questa equazione implica che

$$p = \rho g(\zeta - z)$$

cioè la distribuzione di pressione è idrostatica quindi

$$\frac{\partial p}{\partial x} = \rho g \frac{\partial \zeta}{\partial x}$$

una forma simile vale anche per $\frac{\partial p}{\partial y}$.

Ora è possibile integrare sulla profondità l'equazione di continuità 1.3 da $z = -b$ a $z = \zeta$ poiché sia b sia ζ dipendono da t, x, e y applicando la regola di Leibniz si ottiene

$0 = \int_{-b}^{\zeta} \nabla \cdot \mathbf{v} dz = \int_{-b}^{\zeta} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) dz + w \mid_{z=\zeta} + w \mid_{z=-b} = \frac{\partial}{\partial x} \int_{-b}^{\zeta} u \, dz + \frac{\partial}{\partial y} \int_{-b}^{\zeta} v \, dz - \left(u \mid_{z=\zeta} \frac{\partial \zeta}{\partial x} + u \mid_{z=-b} \frac{\partial b}{\partial x} \right) - \left(v \mid_{z=\zeta} \frac{\partial \zeta}{\partial y} + v \mid_{z=-b} \frac{\partial b}{\partial y} \right) + w \mid_{z=\zeta} + w \mid_{z=-b}$

Definendo le velocità medie come

$$\bar{u} = \frac{1}{H} \int_{-b}^{\zeta} u \, dz$$

$$\bar{v} = \frac{1}{H} \int_{-b}^{\zeta} v \, dz$$

e usando le condizioni al contorno indicate negli elenchi 1. e 2. per eliminare i termini di contorno si trova la seguente equazione di continuità mediata sulla profondità

$$\frac{\partial H}{\partial t} + \frac{\partial}{\partial x} (H\bar{u}) + \frac{\partial}{\partial y} (H\bar{v}) = 0 \tag{1.6}$$

Dopo l'equazione di continuità è necessario integrare anche l'equazione del momento 1.4 per le componenti orizzontali (x e y), su z è già stato fatto per ricavare l'espressione della derivata della pressione. Integrando il membro sinistro dell'equazione della conservazione del momento per l'asse x si ottiene:

$$\int_{-b}^{\zeta} \left[\frac{\partial}{\partial t} u + \frac{\partial}{\partial x} u^2 + \frac{\partial}{\partial y} (uv) + \frac{\partial}{\partial z} (uw) \right] dz =$$

$$= \frac{\partial}{\partial t} (H\bar{u}) + \frac{\partial}{\partial x} \left(H\bar{u}^2 \right) + \frac{\partial}{\partial y} (H\bar{u}\bar{v}) + \{avv\} \tag{1.7}$$

dove $\{avv\}$ sono termini differenziali di avvezione che tengono conto che la media del prodotto di due funzioni non è il prodotto delle medie. Si ritrova un risultato simile anche per l'asse y. Integrando sulla profondità i membri destri di tali equazioni si ottengono

$$-\rho g H + \tau_{sx} - \tau_{bx} + \frac{\partial}{\partial x} \int_{-b}^{\zeta} \tau_{xx} + \frac{\partial}{\partial y} \int_{-b}^{\zeta} \tau_{xy} \tag{1.8}$$

$$-\rho g H + \tau_{sy} - \tau_{by} + \frac{\partial}{\partial x} \int_{-b}^{\zeta} \tau_{xy} + \frac{\partial}{\partial y} \int_{-b}^{\zeta} \tau_{yy} \tag{1.9}$$

Ricombinando tutto quello che è stato ottenuto, cioè l'equazione di continuità 1.6, la 1.7 e la sua controparte per l'asse y per il membro sinistro delle equazioni della conservazione del momento e la 1.8 e la 1.9 per le loro parti destre, si ottengono le equazioni per le acque basse in due dimensioni o 2D-SWE (2D Shallow Water Equations) in forma conservativa

$$\frac{\partial H}{\partial t} + \frac{\partial}{\partial x} (H\bar{u}) + \frac{\partial}{\partial y} (H\bar{v}) = 0 \tag{1.10}$$

$$\frac{\partial}{\partial t}\left(H\bar{u}\right)+\frac{\partial}{\partial x}\left(H\bar{u}^2\right)+\frac{\partial}{\partial y}\left(H\bar{u}\bar{v}\right)=-gH\frac{\partial\zeta}{\partial x}+\frac{1}{\rho}[\tau_{sx}-\tau_{bx}+F_x] \qquad (1.11)$$

$$\frac{\partial}{\partial t}\left(H\bar{v}\right)+\frac{\partial}{\partial y}\left(H\bar{u}\bar{v}\right)+\frac{\partial}{\partial x}\left(H\bar{v}^2\right)=-gH\frac{\partial\zeta}{\partial y}+\frac{1}{\rho}[\tau_{sy}-\tau_{by}+F_y] \qquad (1.12)$$

Lo sforzo di superficie, la frizione sul fondo e le componenti orizzontali F_x ed F_y delle forze agenti devono essere determinate caso per caso.

1.4.1 Applicazioni

I modelli delle acque basse sono stati sviluppati con lo scopo di indagare gli effetti di esplosioni nucleari vicino alla superficie dell'oceano o sott'acqua [Leendertse, 1967].

Uno degli usi delle equazioni delle acque basse è la determinazione dei pattern di circolazione e delle maree massime e minime all'interno di una regione soggetta a forze di marea ai confini (aperti) della regione stessa [Kinnmark, 1996]. Questi pattern di circolazione sono di primario interesse in un modello per determinare il trasporto di differenti specie per applicazioni ambientali. Un'altra importante applicazione sono gli tsunami, onde generate da terremoti sul fondo degli oceani, quando queste onde si si avvicinano a fondali bassi la loro velocità diminuisce e la loro ampiezza aumenta e questo può causare gravi inondazioni [Kawahara and Yoshida, 1978]. Una possibilità interessante per l'uso delle equazioni del modello shallow water è la valutazione per la conversione dell'energia di marea in energia disponibile commercialmente. Una descrizione interessante di vari progetti proposti per l'energia di marea può essere trovata in [Charlier, 1982] (si vedano anche [Count, 1980], e [Gray and Gashaus, 1972]). Un'altra possibile applicazione è la generazione di altezze d'onda che possano essere usate come dati di input per determinare i carichi delle strutture offshore.

Capitolo 2

Metodi di risoluzione

Lo scopo della tesi è una valutazione preliminare di OpenCL per lo sviluppo di modelli numerici, non ci si focalizzerà in modo particolare sul metodo, per tale motivo verrà utilizzato uno schema semplice a volumi finiti di tipo Lax-Friedrichs. Tale metodo è uno schema di risoluzione esplicito di equazioni differenziali alle derivate parziali di tipo iperbolico[Toro, 2001].

In questo capitolo si descriveranno prima una forma generale di legge di conservazione con forme generali per le soluzioni e le condizioni per la stabilità di queste, poi saranno descritti il metodo Lax-Friedrichs e la sua applicazione a questa tesi, riportando anche codice Fortran e C.

2.1 Formulazione generale delle equazioni di conservazione

Qui di seguito si mostrerà come può essere ottenuto uno schema di soluzione a partire da considerazioni su equazioni di conservazione basandosi su [Leveque, 2002].

La più semplice legge di conservazione unidimensioniale è un'equazione alle derivate parziali del seguente tipo

$$q(x,t) + f(q(x,t))_x = 0$$

dove $f(q)$ è una funziona di flusso. Questa equazione può essere riscritta in forma quasilineare

$$q_t + f'(q)q_x = 0$$

Spesso da considerazioni fisiche emergono equazioni in forma integrale di cui l'esempio più semplice è

$$\int_{x_1}^{x_2} q(x,t)dx$$

La quantità q e il suo integrale possono però variare nel tempo dovendo rispettare variazioni di flusso ai bordi dell'area considerata, questo è imposto dalla seguente legge di conservazione:

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)dx = F_1(t) - F_2(t)$$

dove $F_1(t)$ e $F_2(t)$ sono entrambi flussi verso l'area considerata e se queste sono funzioni che dipendono solamente dalla quantità q si può riscrivere la seguente equazione, sempre per il caso unidimensionale, che sarà considerata per il seguito

$$\frac{d}{dt}\int_{x_1}^{x_2} q(x,t)dx = f(q(x_1,t)) - f(q(x_2,t)) \tag{2.1}$$

In una sola dimensione un metodo a volumi finiti consiste nel suddividere il dominio spaziale in intervalli (che sono appunto i "volumi finiti" o *celle della griglia*) e tenere traccia di un'approssimazione dell'integrale di q su ognuno di questi volumi. In ogni passo temporale i valori devono essere aggiornati utilizzando approssimazioni del flusso che attraversa gli estremi dell'intervallo. La i-esima cella può essere denotata con

$$C_i = (x_{i-1/2,i+1/2})$$

Il valore Q_i^n approssima la media sull'intervallo i-esimo a tempo t_n nel seguente modo

$$Q_i^n \approx \frac{1}{\triangle x}\int_{x_{i-1/2}}^{x_{i+1/2}} q(x,t_n)dx \equiv \frac{1}{\triangle x}\int_{C_i} q(x,t_n)$$

dove $\triangle x = x_{i+1/2} - x_{i-1/2}$è la lughezza della cella. Per semplicità si assume una griglia uniforme ma questo non è richiesto.

Se $q(x,t)$ è una funzione regolare allora l'integrale precedente coincide con il valore di q nel punto medio dell'intervallo per $O(\triangle x^2)$. Lavorando con la media delle celle è più facile utilizzare proprietà importanti della legge di conservazione derivando i metodi numerici. In particolare possiamo assicurare che il metodo numerico sia *conservativo* in modo che imiti la vera soluzione.

La sommatoria $\sum_{i=1}^{N} Q_i^n \triangle x$ approssima l'integrale di q sull'intero intervallo $[a,b]$, e se si utilizza il metodo in *forma consevativa* descritto tra breve allora la somma può cambiare solo a causa dei flussi agli estremi $x = a$ ed $x = b$. La massa totale nel dominio computazionale è conservata o varia in accordo alle condizioni di contorno.

La forma integrale della legge di conservazione diventa in questo caso

$$\frac{d}{dt}\int_{C_i} q(x,t)dx = f(q(x_{i-1/2},t)) - f(q(x_{i+1/2},t)) \tag{2.2}$$

Questa espressione può essere usata per sviluppare uno schema esplicito di soluzione. Data la media della cella i al tempo t_n indicata con Q_i^n è possibile

approssimare Q_i^{n+1}, cioè la media al tempo t_{n+1} dopo un passo temporale di $\triangle t = t_{n+1} - t_n$. Integrando l'equazione precedente 2.2 nel tempo da t_n a t_{n+1} si ottiene

$$\int_{C_i} q(x, t_{n+1})dx - \int_{C_i} q(x, t_n)dx = \int_{t_n}^{t_{n+1}} f(q(x_{i-1/2}, t)) - \int_{t_n}^{t_{n+1}} f(q(x_{i+1/2}, t))$$

Riordinando e dividendo per Δx si ottiene

$$\frac{1}{\triangle x}\int_{C_i} q(x, t_{n+1})dx =$$

$$= \frac{1}{\triangle x}\int_{C_i} q(x, t_n)dx - \frac{1}{\triangle x}\left[\int_{t_n}^{t_{n+1}} f(q(x_{i+1/2}, t)) - \int_{t_n}^{t_{n+1}} f(q(x_{i-1/2}, t))\right] \tag{2.3}$$

Questa è una soluzione esatta, infatti indica esattamente come debba essere aggiornata la media in ogni cella ottenendo il valore al passo successivo. In generale però non è possibile vautare esattamene gli integrali al secondo membro dell'equazione poiché $q(x_{i\pm 1/2}, t)$ varia nel tempo lungo il bordo di ogni cella e non si dispone di una soluzione esatta. La formula 2.3 suggerisce comunque che la formuazione generale di un metodo risolutivo sia

$$Q_i^{n+1} = Q_i^n - \frac{\triangle t}{\triangle x}(F_{i+1/2}^n - F_{i-1/2}^n) \tag{2.4}$$

dove $F_{i-1/2}^n$ è una qualche approssimazione del flusso attraverso $x = x_{i-1/2}$

$$F_{i-1/2}^n \approx \frac{1}{\triangle t}\int_{t_n}^{t_{n+1}} f(q(x_{i-1/2}, t))$$

Trovando un modo per approssimare questo flusso basandosi sui valori di Q^n è possibile trovare un metodo per la soluzione.

Per un problema iperbolico l'informazione si propaga ad una velocità finita così è ragionevole che i valori di $F_{i-1/2}^n$ possano essere ottenuti a partire dai valori di Q_{i-1}^n e Q_i^n(si veda la sezione successiva) usando la seguente formula

$$F_{i-1/2}^n = \mathcal{F}(Q_{i-1}^n, Q_i^n)$$

dove $\mathcal{F}$ è una funzione numerica. Il metodo di soluzione 2.4 diventa quindi

$$Q_i^{n+1} = Q_i^n - \frac{\triangle t}{\triangle x}(\mathcal{F}(Q_i^n, Q_{i+1}^n) - \mathcal{F}(Q_{i-1}^n, Q_i^n)) \tag{2.5}$$

Il metodo specifico dipende dalla scelta di F ma in generale un metodo di questo tipo è un metodo esplicito con un *three-point stencil* il che significa che il valore di Q_i^{n+1} dipende solo dai tre valori $Q_{i-1}^n Q_i^n$ e Q_{i+1}^n al passo temporale precedente. Inoltre questi metodi sono detti in *forma conservativa* poiché imitano la proprietà della soluzione esatta.

Si noti che se si effettua la somma $\triangle x Q_{i+1}^n$ dalla formula generale del metodo numerico 2.4 sull'intero insieme di celle si ottiene

$$\triangle x \sum Q^{n+1} = \triangle x \sum Q^n - \frac{\triangle t}{\triangle x}(F_{i+1/2}^n - F_{i-1/2}^n) \tag{2.6}$$

La somma delle differenze di flusso si annulla eccetto per i flussi ai bordi estremi. Sull'intero dominio abbiamo una conservazione esatta eccetto per le condizioni ai bordi. Il metodo di soluzione può essere visto come un'approssimazione diretta a differenza finita alla legge di conservazione $q_t + f(q)_x = 0$ poiché riordinando la 2.6 diventa

$$\frac{Q_i^{n+1} - Q_i^n}{\triangle t} + \frac{F(Q_i^n, Q_{i+1}^n) - F(Q_{i-1}^n, Q_i^n)}{\triangle x} = 0 \tag{2.7}$$

Molti metodi possono essere allo stesso modo visti come approssimazioni a differenze finite o come metodi a volumi finiti.

2.2 La condizione CFL

Courant, Friedrichs e Lewy (da cui la sigla CFL) scrissero i primi articoli relativi ai metodi a differenze finite per equazioni alle derivate parziali nel 1928. Usarono metodi a differenze finite come metodo analitico per provare l'esistenza di soluzioni ad alcune equazioni differenziali alle derivate parziali. L'idea era quella di definire una sequenza di soluzioni approssimate (con equazioni a differenze finite) provando che esse convergono quando la griglia è raffinata e mostrare che il limite deve soddisfare l'equazione fornendo così la soluzione corretta.

CFL indica quindi una condizione necessaria alla stabilità di un metodo a volumi finiti o differenze finite. Questa condizione stabilisce che nel metodo usato l'informazione si propaghi secondo velocità fisiche corrette come determinato dagli autovalori del flusso Jacobiano $f'(q)$.

Con il metodo esplicito riportato nella sezione precedente il valore di Q_i^{n+1} dipende solo dai valori al passo precedente Q_i^n, Q_i^n, Q_{i+1}^n. Nel caso in cui $\bar{u}\triangle t < \triangle x$ l'informazione si propaga meno di una cella in un singolo passo temporale. In questo caso ha senso definire il flusso a $x_{i-1/2}$ in termini dei soli Q_{i-1}^ne Q_i^n. Nel caso in cui $\bar{u}\triangle t > \triangle x$ il flusso $x_{i-1/2}$ dipende anche da Q_{i-2}^n come pure Q_i^{n+1}. Il metodo generale della sezione precedente risulta sicuramente instabile quando applicato con un passo di questo tipo, non importa come sia specificato il flusso se questo dipende solo da Q_{i-1}^ne Q_i^n.

Questa è una conseguenza dalla condizione CFL infatti Courant, Firedrichs e Lewy trovarono la seguente condizione *necessaria* di *stabilità* per ogni metodo numerico

> *Condizione CFL*: Un metodo numerico può convergere solo se il suo dominio numerico di dipendenza contiene il vero dominio di dipendenza dell'equzione differenziale a derivate parziali, almeno nei limti con intervalli temporali e spaziali tendenti a zero

È molto importante notare che la condizione CFL è solo *necessaria* per la *stabilità* (e quindi la *convergenza*), non è sempre *sufficiente* per garantire la stabilità. Nella prossima sezione è riportato un metodo instabile anche se la condizione CFL è rispettata.

Considerando l'equazione di avvezione $q_t + \bar{u}q_x = 0$ con $\bar{u} > 0$, in modo che la soluzione esatta si sposti semplicemente alla velocita $\bar{u}$ e si propaghi alla distanza $\bar{u}\triangle t$ in un passo temporale, la condizione CFL per una soluzione di tipo *three-point stencil* può essere riportata a

$$\nu \equiv |\frac{\bar{u}\triangle t}{\triangle x}| \leq 1 \tag{2.8}$$

dove ν è chiamato a volte numero CFL o, più spesso, numero di Courant (si veda [Leveque, 2002] per una spiegazione di come possa essere ottenuta). Tale numero misura la frazione di cella della griglia che l'informazione propaga in un passo temporale. La condizione può essere meno restrittiva se lo *stencil* è più ampio, con un metodo *five-point stencil* la condizione diventa ≤ 2. Per i problemi iperbolici come le equazioni delle acque basse si usano tipicamente metodi esplici per cui il numero di Courant è minore di 1. Questo consente di mantenere$\triangle t/\triangle x$ fisso raffinando la griglia, questo consente di incrementare la risoluzione alla stesso ritmo sia nel tempo sia nello spazio.

2.3 Un flusso instabile

Come già riportato nella sezione 2.1 il flusso da considerare per risolvere con un metodo a volumi finiti un sistema iperbolico può essere definito in vari modi. Considerando il flusso definito con una funzione F che dipenda solo da Q^n_{i-1} e Q^n_i possiamo come primo tentativo usare una semplice media

$$F^n_{i-1/2} = \mathcal{F}(Q^n_{i-1}, Q^n_i) = \frac{1}{2}[f(Q^n_{i-1}) + f(Q^n_i)]$$

Utilizzando questa definizione nella formulazione generale delle soluzioni otteniamo

$$Q^{n+1}_i = Q^n_i - \frac{\triangle t}{2\triangle x}[f(Q^n_{i+1}) - f(Q^n_{i-1})]$$

Sfortunatamente questo metodo è in generale instabile per problemi iperbolici e non può essere usato nemmeno se il passo temporale è abbastanza piccolo da rispettare la condizione CFL.

2.4 Il metodo Lax-Friedrichs

Il metodo Lax-Friedrichs classico ha la forma

$$Q^{n+1}_i = \frac{1}{2}(Q^n_{i-1} + Q^n_{i+1}) - \frac{\triangle t}{2\triangle x}(f(Q^n_{i+1}) - f(Q^n_{i-1}) \tag{2.9}$$

Questo è molto simile al metodo instabile citato nella sezione precedente ma il valore Q_i^n è sostituito dalla media $\frac{1}{2}(Q_{i-1}^n + Q_{i+1}^n)$. Per un'equazone iperbolica lineare questo metodo è stabile se $\nu \leq 1$ dove il numero di Courant ν è definito come in 2.8.

Ad un primo sguardo tale metodo non appare del tipo delle soluzioni generali 2.5 citate nella sezione 2.1 tuttavia l'equazione può essere ricondotta a quella forma definendo il flusso come

$$\mathcal{F}(Q_{i-1}^n, Q_i^n) = \frac{1}{2}[f(Q_{i-1}^n) + f(Q_i^n)] - \frac{\triangle x}{2\triangle t}(Q_i^n - Q_{i-1}^n) \tag{2.10}$$

Si noti che questo metodo somiglia al flusso instabile centrato con l'aggiunta di un termine simile al flusso dell'equazione di diffusione (si vedano i paragrafi 4.2 e 4.6 di [Leveque, 2002] per ulteriori dettagli). Questo termine può essere interpetato come una *diffusione numerica* che smorza le instabilità e fronisce un metodo che è stabile per numeri di Courant fino a 1 (che è anche il limite CFL per questo metodo a tre punti). Il metodo Lax-Friedrichs ha alcuni svantaggi, infatti introduce più diffusione di quela effettivamente richiesta e fornisce risultati che sono tipicamente mal distribuiti a meno che non venga utilizzata una griglia molto fine.

2.5 Il metodo Lax-Friedrichs per le acque basse

Nel caso delle acque basse ci sono due differenze con il caso generale riportato, la prima è la bidimensionalità del dominio, la seconda è che la soluzione che di interesse non è la quantità media di fluido in ogni cella ma l'altezza del fluido in quella cella. Indicando con Z l'altezza dell'acqua in una cella si può ottenere il valore dall'altezza della cella (i, j) al passo $n+1$ utilizzando solamente i valori di Z e Q al passo n adattando il metodo di Lax-Friedrichs esposto precedentemente nel seguente modo

$$Z_{i,j}^{n+1} =$$

$$= \frac{1}{2}(Z_{i+1,j}^n + Z_{i-1}^n + Z_{i,j+1}^n + Z_{i,j-1}^n) - \frac{\triangle t}{2\triangle x}(Qx_{i+1,j}^n - Qx_{i-1,j}^n + Qy_{i,,j+1}^n - Qy_{i,j-1}^n) \tag{2.11}$$

Come si può notare la media all'inizio non è più sui valori di Q ma su quelli di Z delle 4 celle adiacenti (questo caso è bidimensionale), mentre il resto è un termine che dipende ancora delle medie di cella Q (che ora sono distinte in Qx e Qy per differenziare i contributi e la conservazione sui due assi) del passo precedente.

2.6 Codice Fortran e C

Il primo passo è stato tradurre il codice Fotran (che disponeva di un output su file con formato leggibile dal programma *plotmtv*) in linguaggio C (si è in seguito

deciso di convertire l'output in un formato leggibile dal programma *Paraview*) usando l'approccio più semplice possibile implementando l'algoritmo in maniera sequenziale, senza quindi introdurre concorrenza né parallelismo e usando solo le caratteristiche del linguaggio C e delle relative librerie standard, senza usare perciò le funzioni offerte dal sistema operativo né librerie esterne.

Qui di seguito si riporta la parte centrale del codice in Fortran nel caso di bordo chiuso:

Algoritmo 2.1 Codice Fortran per il Metodo Lax-Friedrichs per la simulazione delle acque basse

```
1  Qx_old(0,:)=-Qx_old(1,:)
2  Qx_old(Nx+1,:)=-Qx_old(Nx,:)
3  Qx_old(:,0)=Qx_old(:,1)
4  Qx_old(:,Ny+1)=Qx_old(:,Ny)
5  Qy_old(0,:)=Qy_old(1,:)
6  Qy_old(Nx+1,:)=Qy_old(Nx,:)
7  Qy_old(:,0)=-Qy_old(:,1)
8  Qy_old(:,Ny+1)=-Qy_old(:,Ny)
9
10 do i=1,Nx
11   do j=1,Ny
12     Z(i,j)=0.25d0*(Z_old(i+1,j)+Z_old(i-1,j)+Z_old(i,j+1)+
           Z_old(i,j-1))-dt/(2.d0*dx)*(Qx_old(i+1,j)-Qx_old(i-1,
           j)+Qy_old(i,j+1)-Qy_old(i,j-1))
13     mass=mass+Z(i,j)*(dx**2)
14     Fpx=Qx_old(i+1,j)**2/Z_old(i+1,j)+0.5*9.8*Z_old(i+1,j)
15     Fmx=Qx_old(i-1,j)**2/Z_old(i-1,j)+0.5*9.8*Z_old(i-1,j)
16     Gpx=Qx_old(i+1,j)*Qy_old(i+1,j)/Z_old(i+1,j)
17     Gmx=Qx_old(i-1,j)*Qy_old(i-1,j)/Z_old(i-1,j)
18     Qx(i,j)=0.25d0*(Qx_old(i+1,j)+Qx_old(i-1,j)+Qx_old(i,j
           +1)+Qx_old(i,j-1))-dt/(2.d0*dx)*(Fpx-Fmx+Gpx-Gmx)
19     Fpy=Qx_old(i,j+1)*Qy_old(i,j+1)/Z_old(i,j+1)
20     Fmy=Qx_old(i,j-1)*Qy_old(i,j-1)/Z_old(i,j-1)
21     Gpy=Qy_old(i,j+1)**2/Z_old(i,j+1)+0.5*9.8*Z_old(i,j+1)
22     Gmy=Qy_old(i,j-1)**2/Z_old(i,j-1)+0.5*9.8*Z_old(i,j-1)
23     ! Computation of friction term along y
24     u=Qx(i,j)/Z(i,j)
25     v=Qy(i,j)/Z(i,j)
26     S=(n**2)*v*dsqrt(u**2+v**2)/(Z(i,j)**(4.d0/3.d0))
27     Qy(i,j)=0.25d0*(Qy_old(i+1,j)+Qy_old(i-1,j)+Qy_old(i,j
           +1)+Qy_old(i,j-1))-dt/(2.d0*dx)*(Fpy-Fmy+Gpy-Gmy)
28   end do
29 end do
```

Si fa notare che il doppio asterisco corrisponde all'operazione di elevamento a potenza. Si aggiunge che questa è solo la parte centrale del codice che infatti comprende una parte di inizializzazione (che include anche la scelta dei valori di

discretizzazione e il controllo che soddisfino la condizione CFL) e le istruzioni per l'output su file. Si fa notare anche che i due cicli innestati sono riferiti solo alla discretizzazione nello spazio infatti questo blocco di codice è eseguito in un ciclo esterno che corrisponde all'andamento temporale della simulazione e viene eseguito per un numero di passi pari a `nstep = ceil(Time/dt)` dove *Time* è stato scelto dall'utente e *dt* calcolato come `dt = dx/sqrt(9.8*(Zmax-Zmin)) *0.1` oppure inserito dall'utente se ha scelto di cambiarlo.

Qui di seguito invece si riporta il corrispettivo in linguaggio C:

Algoritmo 2.2 Codice C per il Metodo Lax-Friedrichs per la simulazione delle acque basse

```
1  copyrow(Z_old, 0, Z_old, 1, Ny+2);
2  copyrow(Z_old, Nx+1, Z_old, Nx, Ny+2);
3  copycolumn(Z_old, 0, Z_old, 1, Nx+2);
4  copycolumn(Z_old, Ny+1, Z_old, Ny, Nx+2);
5  copyinverserow(Qx_old, 0, Qx_old, 1, Ny+2);
6  copyinverserow(Qx_old, Nx+1, Qx_old, Nx, Ny+2);
7  copycolumn(Qx_old, 0, Qx_old, 1, Nx+2);
8  copycolumn(Qx_old, Ny+1, Qx_old, Ny, Nx+2);
9  copyrow(Qy_old, 0, Qy_old, 1, Ny+2);
10 copyrow(Qy_old, Nx+1, Qy_old, Nx, Ny+2);
11 copyinversecolumn(Qy_old, 0, Qy_old, 1, Nx+2);
12 copyinversecolumn(Qy_old, Ny+1, Qy_old, Ny, Nx+2);
13
14 for(i=1; i<=Nx; i++)
15 {
16   for(j=1; j<=Ny; j++)
17   {
18             Z[i][j]=0.25*(Z_old[i+1][j]+Z_old[i-1][j]+Z_old[
                   i][j+1]+Z_old[i][j-1])-dt/(2.0*dx)*(Qx_old[i
                   +1][j]-Qx_old[i-1][j]+Qy_old[i][j+1]-Qy_old[i
                   ][j-1]);
19     if(usemass)
20     {
21       mass=mass+Z[i][j]*dx*dx;
22     }
23     Fpx=Qx_old[i+1][j]*Qx_old[i+1][j]/Z_old[i+1][j]+0.5*9.8*
           Z_old[i+1][j];
24     Fmx=Qx_old[i-1][j]*Qx_old[i-1][j]/Z_old[i-1][j]+0.5*9.8*
           Z_old[i-1][j];
25     Gpx=Qx_old[i+1][j]*Qy_old[i+1][j]/Z_old[i+1][j];
26     Gmx=Qx_old[i-1][j]*Qy_old[i-1][j]/Z_old[i-1][j];
27     Qx[i][j]=0.25*(Qx_old[i+1][j]+Qx_old[i-1][j]+Qx_old[i][j
           +1]+Qx_old[i][j-1])-dt/(2*dx)*(Fpx-Fmx+Gpx-Gmx);
28     Fpy=Qx_old[i][j+1]*Qy_old[i][j+1]/Z_old[i][j+1];
29     Fmy=Qx_old[i][j-1]*Qy_old[i][j-1]/Z_old[i][j-1];
30     Gpy=Qy_old[i][j+1]*Qy_old[i][j+1]/Z_old[i][j
           +1]+0.5*9.8*(Z_old[i][j+1]);
31     Gmy=Qy_old[i][j-1]*Qy_old[i][j-1]/Z_old[i][j
           -1]+0.5*9.8*(Z_old[i][j-1]);
32     /*computation of friction term along y*/
33     u=Qx[i][j]/Z[i][j];
34     v=Qy[i][j]/Z[i][j];
35     S=(n*n)*v*sqrt(u*u+v*v)/pow(Z[i][j],4.0/3.0);
36     Qy[i][j]=0.25*(Qy_old[i+1][j]+Qy_old[i-1][j]+Qy_old[i][j
           +1]+Qy_old[i][j-1])-dt/(2*dx)*(Fpy-Fmy+Gpy-Gmy);
37   }
38 }
```

Si noti la variabile *usemass* che indica se l'utente vuole tenere conto della massa o no che nel caso sia falsa evita l'aggiornamento della massa; questo sarà importante nell'implementazione OpenCL.

L'unica caratteristica che ha reso la traduzione realmente un po' più di una semplice traduzione sintattica cambiando le parole chiave e la notazione dei blocchi è stata la gestione delle matrici, il Fortran usa costrutti semplici per l'allocazione dinamica di matrici e la copia di righe o colonne e il programma in Fortran utilizza questi costrutti per allocare matrici la cui dimensione è scelta sulla base dei dati di input, non è quindi possibile sceglierla al tempo di compilazione.

Per la rappresentazione di matrici si è scelto quindi di utilizzare variabili di tipo puntatore a puntatore (che possono indirizzare aree di memoria allocate dinamicamente con la funzione malloc) che sono stati poi interpretati come array di array e quindi matrici il cui accesso è stato fatto con la normale notazione a doppio indice tra parentesi quadre (un'altra scelta sarebbe potuta essere l'utilizzo di un solo puntatore interpretato come vettore e solo a livello più alto come matrice, come indicato nel capitolo 4 questo approccio è stato usato per l'implementazione con OpenCL). In conclusione si sono create funzioni dedicate per l'allocazione delle matrici sfruttando puntatori a puntatori e funzioni dedicate per la copia di righe e colonne (invocate all'inizio del blocco di codice riportato) che oltre alle variabili in oggetto e alla riga o colonna da copiare richiedono tra i parametri anche la lunghezza della stessa (che corrisponde al numero di colonne per la riga e al numero di righe per la colonna) poiché non è un'informazione contenuta nella variabile stessa come si sarebbe potuto fare utilizzando dei tipi complessi come quelli struttura costruibili dal linguaggio C o gli oggeti in un linguaggio che li supporta.

Al contrario del codice originale in Fortran che produceva output su file leggibile da *plotmv* l'implementazione C utilizza il formato VTK riconosciuto da paraview. Qui di seguito un esempio, il simbolo [...] indica una sezione omessa, in particolare una sequenza di valori ordinati facilmente interpretabile.

```
# vtk DataFile Version 1.0 Shallow water surface at
    time 10
ASCII
DATASET RECTILINEAR_GRID
DIMENSIONS 101 101 1
X_COORDINATES 101 float
0.000000000000000 5.000000000000000 [...]
    495.000000000000000 500.000000000000000
Y_COORDINATES 101 float
0.000000000000000 5.000000000000000 [...]
    495.000000000000000 500.000000000000000
Z_COORDINATES 1 float
0.000000000000000
CELL_DATA 10000
SCALARS depth float
```

```
LOOKUP_TABLE default
[qui sono presenti 10000 valori corrispondenti alle
   profondità associate celle]
VECTORS velocities float
[qui sono presenti 10000 terne corrispondenti alle
   profondità associate alle celle, le terne
   rappresentano vettori nello spazio ma il terzo
   valore è sempre 0.000000000000000 poiché il modello
    è bidimensionale e i vettori sono quindi
   orizzontali rispetto al piano di fondo]
```

Il numero di coordinate elencate è 101 per ogni dimensione e i valori sono 10000 poiché le coordinate indicano i bordi delle celle, i dati sono invece associato al centro e quindi è necessaria una coordinata in più rispetto al numero di celle.

Il codice soffre di alcune limitazioni: non gestisce livelli del fluido pari a zero quindi non può ad esempio simulare la rottura di una diga verso una zona asciutta o uno straripamento (è comunque sempre possibile utilizzare valori molto prossimi allo zero per casi di questo tipo, un approccio migliore sarebbe tuttavia poter utilizzare il valore zero ed evitare i calcoli sule celle corrispondenti a tale valore) e presume il fondo piatto.

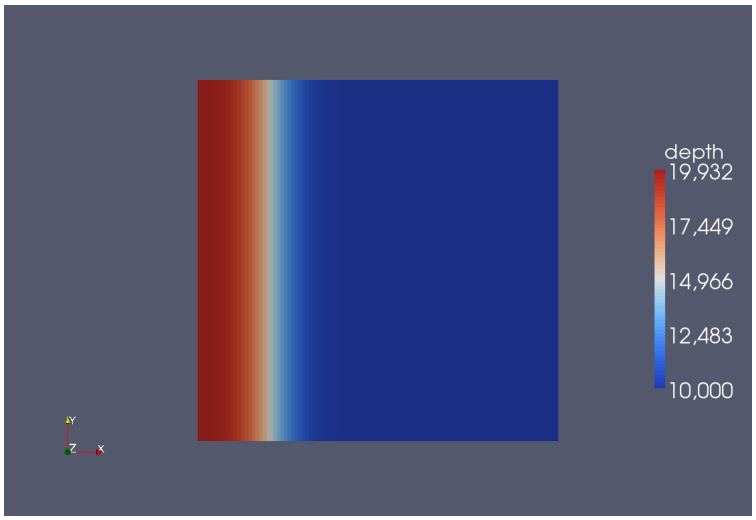


Figura 2.1: Esempio di output Paraview per una simulazione in doppia precisione di dimensione 100x100 e discretizzazione 5 al passo 100 (t=5.0507627227611≈5.051) di 396. Nella legenda dell'immagine la virgola è usata come separatore della parte frazionaria.

Capitolo 3

Calcolo parallelo, GPU computing e OpenCL

In questo capitolo dopo una breve introduzione riguardante il calcolo eterogeneo si descriveranno il concetto e la storia del calcolo parallelo, il concetto e la storia del GPU computing con le piattaforme di ATI e NVIDIA per il suo utilizzo compresi esempi di applicazione (soprattutto di CUDA C per le GPU Nvidia con architettura CUDA), la situazione attuale e futura del calcolo eterogeneo, lo standard OpenCL per il calcolo parallelo, i diversi modelli di calcolo parallelo utilizzabili in OpenCL e le peculiarità di CPU e GPU (entrambe sono utilizzabili, con OpenCL o altri metodi, per eseguire calcoli in parallelo ma ognuna ha una specifica architettura sfruttabile al meglio con specifiche scelte di implementazione) fornendo anche come esempio l'architettura di una GPU NVIDIA e quella di una GPU AMD.

3.1 Introduzione al calcolo eterogeneo

Le attività recenti dei produttori di processori e GPU come NVIDIA, AMD (considerando sia i processori sia le GPU ATI) e Intel rendono evidente più che mai che i progetti futuri di microprocessori e sistemi HPC (High-Performance Computing) avranno una natura ibrida ed eterogenea. Questi sistemi eterogenei si affideranno all'integrazione di due tipi di componenti in diverse proporzioni:

- Tecnologia multi-core e many-core per le CPU: il numero di core continua ad aumentare a causa del desiderio di inserire sempre più componenti su un singolo chip per evitare di accrescere costantemente la potenza assorbita, il parallelismo a livello di istruzione e la quantità di memoria richiesta.
- Hardware specializzato e acceleratori massicciamente paralleli: le GPU fornite da NVIDIA e AMD/ATI hanno superato le performance delle CPU nei calcoli in virgola mobile negli ultimi anni, sono inoltre diventate molto più facili da programmare

Il bilancio relativo tra questi tipi di componenti non è chiaro e continuerà a variare nel tempo, non c'è però dubbio che i sistemi futuri (dai laptop fino ai supercomputer) saranno composti da componenti eterogenei infatti la barriera del petaflop (10^{15}) è stata superata da uno di questi sistemi.

I problemi e le sfide che gli sviluppatori devono affrontare nel nuovo campo dei processori ibridi rimagono preoccupanti. Parti critiche dell'infrastruttura software stanno già avendo difficoltà ad essere tenute al passo coi cambiamenti. In alcuni casi le performance non riescono a scalare adeguatamente con il numero di core poiché la maggior parte del tempo di esecuzione dell'applicazione è speso per spostare dati piuttosto che per effettuare calcoli aritmetici. In altri casi il codice ottimizzato per le performance su un determinato hardware è consegnato anni dopo l'arrivo di quell'hardware che nel frattempo è diventato obsoleto, il software risulta così obsoleto già alla consegna. In alcuni casi (come per GPU più recenti) il software non può essere eseguito poiché l'ambiente di programmazione è cambiato troppo. Questa introduzione è basata sulla prefazione di [Sanders and Kandrot, 2010], la storia del calcolo parallelo, del GPU computing e la descrizione di CUDA sono basati soprattutto sul primo capitolo dello stesso.

3.2 Storia del calcolo parallelo con CPU

Tempo fa la programmazione parallela era considerata un argomento "esotico" ed era considerata un argomento particolare dell'informatica. Questa percezione è cambiata profondamente nel corso degli anni, ora quasi ogni programmatore necessita di qualche nozione relativa a questo argomento.

Ormai da molto quasi tutti i computer venduti equipaggiano più core (dai due di alcuni netbook e personal computer di tipo desktop agli 8 o 16 di server e workstation) e la programmazione parallela non è più relegata ai supercomputer o ai mainframe. Persino alcuni telefoni e lettori multimediali incorporano la possibilità di effettuare calcoli paralleli per poter fornire funzionalità maggiori rispetto ai predecessori.

Tuttavia gli sviluppatori devono gestire diversi tipi di piattaforme e tecnologie per poter offrire esperienze ricche e innovative per basi di utenti sempre più sofisticate ed esigenti, non basta più offrire un'interfaccia grafica accattivante e semplice da usare in ogni applicazione, persino i telefoni cellulari devono essere dispositivi in grado di gestire più applicazioni grafiche contemporaneamente come ad esempio un Browser Web, un riproduttore musicale, un'applicazione che trovi informazioni sulla base del GPS del dispositivo ed eventualmente avviare una telefonata senza che queste applicazioni perdano il loro stato.

Per trent'anni uno dei metodi importanti per aumentare le performance dei dispositivi di calcolo per il mercato domestico è stato l'aumento della velocità del clock del processore. I primi personal computer degli anni '80 avevano frequenza di clock di circa 1MHz. Trent'anni dopo (2010) la maggior parte dei processori nello stesso mercato ha clock tra 1GHz e 4GHz, il clock è quindi almento 1000 volte più veloce dei primi personal computer. Nonostante la maggior velocità del

clock non sia certamente l'unico modo per aumentare le performance è sempre stata una fonte affidabile per il miglioramento delle prestazioni.

Negli anni recenti comunque i produttori di hardware sono stati forzati a cercare alternative per aumentare la potenza di calcolo. A causa di varie limitazioni fondamentali nella fabbricazione di circuiti integrati non è più possibile affidarsi alla spirale verso l'alto delle frequeze di clock dei processori per estrarre maggior potenza di calcolo dalle architetture esistenti. A causa di limitazioni di assorbimento di potenza e rilascio di calore e a causa dell'avvicinamento al limite fisico della dimensione dei transistor ricercatori e produttori hanno iniziato a a guardare altrove.

Al di fuori del mercato domestico i supercomputer hanno per decenni ottenuto enormi guadagni di prestazione in modi simili. Le performance dei processori usati nei supercomputer sono aumentate straordinariamente, in modo simile a quello in cui sono aumentate quelle delle CPU dei personal computer. Tuttavia in aggiunta a questo metodo i costruttori di supercomputer hanno fatto enormi balzi di prestazione aumentando costantemente il *numero* dei processori. Non è insolito per i supercomputer avere decine o centinaia di migliaia di processori che lavorano parallelamente.

All'interno della ricerca per ottenere ulteriore potenza di calcolo per i personal computer il miglioramento ottenuto per i supercomputer solleva una domanda. Piuttosto che cercare solamente di di aumentare le performance di un singolo core di elaborazione perché non metterne di più in un personal computer? In questo modo i personal computer possono accrescere le prestazioni senza bisogno di aumentare continuamente la velocità di clock del processore.

Nel 2005 con un mercato sempre più competitivo e poche alternative i produttori di CPU iniziarono a offrire processori con due unità di elaborazione (core) invece di una. Negli anni successivi sono stati seguiti dal rilascio di CPU a tre, quattro e sei core. Questo andamento, a volte indicato come *rivoluzione multi-core* ha segnato uno spostamento nell'evoluzione del mercato domestico dei computer.

Oggi persino i processori di fascia bassa e a basso consumo hanno due o più core per singolo chip e i maggiori produttori di CPU hanno già annunciato piani per CPU a 12 e 16 core, questo conferma che il calcolo parallelo è definitamente arrivato. Il centro di ricerca Intel ha persino sviluppato un processore sperimentale a 80 core per studiare i problemi di carico e comunicazione tra diversi core in un processore così fortemente parallelizzato.

La parallelizzazione dei problemi non può comunque essere sempre applicata e i miglioramenti che ci si possono aspettare dipendono da quali parti di un programma possono essere divise ed eseguite in parallelo, la legge di Amdahl stabilisce che il massimo miglioramento ottenibile in un programma dove P è la percetuale parallelizzabile e N il numero di microprocessori è:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Se ad esempio la parte del codice parallelizzabile è del 60% e si hanno a disposizione 4 processori il miglioramento sarà di $\frac{1}{(1-0.6)+\frac{0.6}{4}} = \frac{1}{\frac{2}{5}+\frac{3}{20}} = \frac{20}{11}$ e

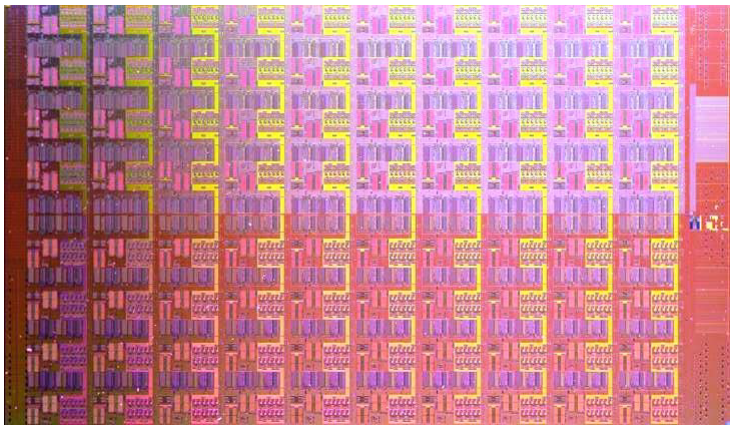


Figura 3.1: Il processore Intel a 80 core

si potrà ottenere al massimo un codice che impiegherà un tempo pari a $\frac{1}{\frac{20}{11}} = \frac{11}{20} = 0.55$ cioè il 55% di quello non parallelizzato. Il perché appare chiaro, il 60% del tempo può essere ridotto con 4 processori al massimo al 15% del tempo totale della versione sequenziale mentre il 40% restante rimane inalterato. Si nota poi che anche disponendo idealmente di un numero infinito di processori considerando il limite per N tendente all'infinito la formula si riduce a $S = \frac{1}{1-P}$ comportando comunque un limite invalicabile. Nell'esempio precedente il miglioramento sarebbe di $\frac{1}{\frac{2}{5}} = \frac{5}{2}$ e il tempo di esecuzione pari a $\frac{2}{5} = 0.4$ cioè il 40% di quello originale questo perché il tempo parallelizzabile si ridurrebbe a zero e rimarrebbe solo il restante 40% sequenziale.

3.3 GPU computing

In confronto alla tradizionale pipeline di elaborazione dei dati del processore centrale l'utilizzo delle GPU per eseguire calcoli general-purpose è un concetto piuttosto nuovo. In effetti le stesse GPU sono recenti considerando l'intera storia dell'informatica. Comunque l'idea di usare le GPU per calcoli general-purpose è nata poco dopo il rilascio delle GPU.

3.3.1 Breve storia delle GPU

Alla fine degli anni '80 e all'inizio degli anni '90 la crescita di popolarità dei sistemi operativi comandati graficamente come Microsoft Windows crearono un mercato per un nuovo tipo di processori. All'inizio degli anni '90 gli utenti iniziarono ad acquistare acceleratori 2D per i loro personal computer. Questi acceleratori offrivano operazioni hardware sulle bitmap per migliorare la visualizzazione e l'usabilità del sistema operativo.

Nello stesso periodo, nel mondo professionale dell'informatica, la Silicon Graphics aveva passato gli anni '80 a rendere popolare l'uso della grafica tridimensionale in diversi mercati, tra cui: applicazioni di difesa e governative, visualizzazioni tecniche e scientifiche e strumenti per creare effetti cinematografici sbalorditivi. Nel 1992 la Silicon Graphics aprì l'interfaccia di programmazione per il suo hardware rilasciando la libreria OpenGL affinché potesse essere utilizzata come uno standard indipendente dalla piattaforma per la scrittura di applicazioni grafiche 3D. Come per il calcolo parallelo e le CPU era solo questione di tempo prima che queste tecnologie raggiungessero gli utenti comuni.

Nella metà degli anni '90 la domanda per applicazioni consumer che impiegassero grafica 3D era cresciuta velocemente, creando la base per due sviluppi significativi.

Primo il rilascio di sparatutto in prima persona immersivi cone Doom, Duke Nukem 3D e Quake iniziarono la corsa alla creazione di ambienti 3D per giochi sempre più realistici. Diversi videogiochi di questo e altri generi accelerarono l'adozione della grafica 3D nel mercato domestico. Secondo, nello stesso tempo compagnie come NVIDIA, ATI e 3dfx Interactive (chiusa per bancarotta nel 2002) iniziarono a rilasciare acceleratori grafici abbastanza accessibii da attirare l'attenzione del grande pubblico.

Questi sviluppi cementarono la grafica 3D come una tecnologia prevalente negli anni successivi.

Il rilascio da parte di NVIDIA della GPU GeForce 256 nel 1999 spinse ancora oltre le capacità dell'hardware grafico casalingo. Per la prima volta calcoli di trasformazione e illuminazione (transform & lighting) potevano essere eseguiti direttamente sul processore grafico aumentando il potenziale per creare applicazioni visuali ancora più interessanti. Poiché trasformazione e illuminazione erano erano già parte integrante della pipeline grafica OpenGL, la GeForce 256 segnò l'inizio di una naturale progressione dove sempre più pipeline sarebbero state implementate direttamente sul processore grafico.

Dal punto di vista del calcolo parallelo il rilascio da parte di NVIDIA della serie GeForce 3 nel 2001 rappresenta probabilmente un punto di svolta nella tecnologia delle GPU. Questa GPU fu il primo chip a implementare l'allora nuovo standard DirectX 8.0. Questo standard richiedeva che l'hardware conforme contenesse vertex shader e pixel shader programmabili. Per la prima volta gli sviluppatori avevano il controllo sugli esatti calcoli che avrebbe fatto la GPU.

3.3.2 GPU computing primordiale

Il rilascio di GPU con pipeline programmabili attirarono molti ricercatori verso la possibilità di usare l'hardware grafico per fare più di semplici rendering OpenGL o DirectX. L'approccio generale in quegli anni era estremamente contorto. Poiché le API grafiche standard OpenGL e DirectX erano ancora gli unici modi di interagire con la GPU ogni tentativo di eseguire calcoli arbitrari sulla GPU era soggetto alle restrizioni di programmazione di quelle API. A causa di questo i ricercatori esplorarono i calcoli general-purpose facendo in modo che i loro problemi apparissero alla GPU come problemi di rendering tradizonali.

Essenzialmente le GPU dei primi anni 2000 erano progettate per produrre un colore per ogni pixel sullo schermo usando unità aritemtiche programmabili chiamate pixel shader. In generale un pixel shader usa la sua posizione in coordinate cartesiane (x,y) sullo schermo insieme a varie altre informazioni per combinare diversi input e calcolare un colore finale. Le informazioni addizionali possono essere i colori di input, le coordinate delle texture o altri attributi passati come parametri allo shader quando è eseguito. Poiché l'aritmetica eseguita sui colori di input e le texture è completamente controllata dal programamtore i ricercatori osservarono che questi "colori" di input potevano in realtà essere *qualunque* dato.

In questo modo se gli input fossero stati effettivamente dati numerici con un significato diverso dal colore i programmatori avrebbero potuto programmare i pixel shader per eseguire calcoli arbitrari su questi dati. Il risultato restituito dalla GPU come "colore" finale del pixel sarebbe stato in realtà il risultato desiderato che avrebbe potuto essere letto dai ricercatori ma la GPU non ne sarebbe stata al corrente. In sostanza la GPU veniva ingannata facendo apparire compiti generali come compiti di rendering, questo inganno era intelligente ma anche molto contorto.

A causa dell'alto throughput aritmetico delle GPU i risultati iniziali di questi esperimenti promettevano un brillante futuro per il GPU computing. Comunque il modello di programmazione era ancora troppo restrittivo perché si formasse una massa critica di sviluppatori. C'erano stretti limiti di memoria poiché i programmi potevano ricevere come dati di input solo una manciata di colori di input e di unità di texture. C'erano serie limitazioni anche su come e dove il programmatore potesse scrivere i risultati nella memoria, perciò gli algoritmi che richiedono di scrivere in locazioni arbitrarie di memoria (scatter) non potevano essere eseguti sulla GPU. Inoltre era quasi impossibile prevedere come una particolare GPU avrebbe trattato i dati in virgola mobile (sempre che fosse stata in grado di trattarli) così molti calcoli scientifici non avrebbero potuto usare la GPU. Infine quando il programma avesse calcolato dati sbagliati, avesse terminato o avesse bloccato la macchina non ci sarebbero stati metodi ragionevolmente buoni per fare il debug del codice eseguto sulla GPU.

Oltra a queste difficoltà chiunque avesse voluto comunque tentare di effetutare calcoli generali su GPU avrebbe dovuto imparare le librerie OpenGL oppure DirectX che erano gli unici modi per comunicare con la GPU. Questo non significava solamente rappresentare i propri dati come texture grafiche e invocare le funzioni DirectX o OpenGL per l'esecuzione ma anche scrivere i calcoli desiderati in linguaggi di programmazione dedicati solo alla grafica chiamati shading language. Chiedere ai ricercatori di gestire severe restrizioni di risorse e programmazione e imparare la computer grafica e i linguaggi di shading ancora prima di tentare di sfruttare la potenza di calcolo delle loro GPU si rivelò un ostacolo troppo grande per un'ampia diffusione.

3.3.3 CUDA

Nel 2006 NVIDIA presentò la prima GPU a supportare le DirectX 10, la GeForce 8800GTX era anche la prima GPU ad utilizzare l'architettura CUDA. Questa architettura includeva vari nuovi componenti progettati specificatamente per il GPU computing e mirava ad eliminare le limitazioni che prevenivano che le GPU precedenti fossero utilizzate per calcoli non grafici.

3.3.3.1 Cos'è CUDA

Diversamente dalle precedenti generazioni che dividevano le risorse di calcolo in vertex e pixel shader l'architettura CUDA (che significa appunto Compute Unified Device Architecture) includeva una shader pipeline unificata, permettendo ad ogni ALU sul chip di essere utilizzata da un programma per effettuare calcoli generali. Poiché NVIDIA intendeva spingere l'utilizzo di queste schede nel settore del GPU computing costruì le ALU in modo che rispettassero gli standard IEEE per i calcoli in virgola mobile a singla precisione e furono progettate per usare un insieme di istruzioni ritagliato per calcoli generici piuttosto che specificatamente per la grafica. Inoltre, alle unità di esecuzione sulla GPU fu permesso di leggere e scrivere arbitrariamente in memoria come anche l'accesso a una cache gestita a livello software chiamata *shared memory*. Tutte queste caratteristiche dell'architettura CUDA furono aggiunte per realizzare una GPU che eccellesse anche in calcoli di tipo generale oltre che nei tradizionali compiti grafici.

3.3.3.2 Utilizzo di CUDA

Lo sforzo di NVIDIA di fornire ai consumatori un prodotto sia per il calcolo sia per la grafica non poteva fermarsi alla sola produzione di hardware comprendente l'architettura CUDA. Indipendentemente dal numero di caratteristiche implementate nei suoi chip per facilitare la computazione non c'era modo che potessero essere utilizzate senza utilizzare le OpenGL o le DirectX. Non solo era ancora richiesto che gli utenti facessero apparire i problemi come calcoli grafici ma dovevano continuare a scrivere i loro programmi utilizzando i linguaggi di shading GLSL (OpenGL Shading Language) o l'HLSL (High Level Shading Language) di Microsoft.

Per raggiungere il massimo possibile di sviluppatori NVIDIA prese il linguaggio C (standard di fatto nell'industria) e ci aggiunse un piccolo numero di parole chiave per poter sfruttare le caratteristiche speciali della GeForce 8800 GTX, NVIDIA rese pubblico il compilatore per questo linguaggio, CUDA C (o anche C for CUDA). CUDA C fu quindi il primo linguaggio progettato espressamente da un produttore di GPU per semplificare l'effettuazione di calcoli general-purpose sulle GPU.

In aggiunta alla creazione di un linguaggio per scrivere codice per la CPU, NVDIA ha anche fornito un driver hardware specializzato per sfruttare l'enorme potenza computazionale dell'architettura CUDA. Questo segnò l'inizio di un nuovo modo di fare GPU computing che non richiedeva più né OpenGL né

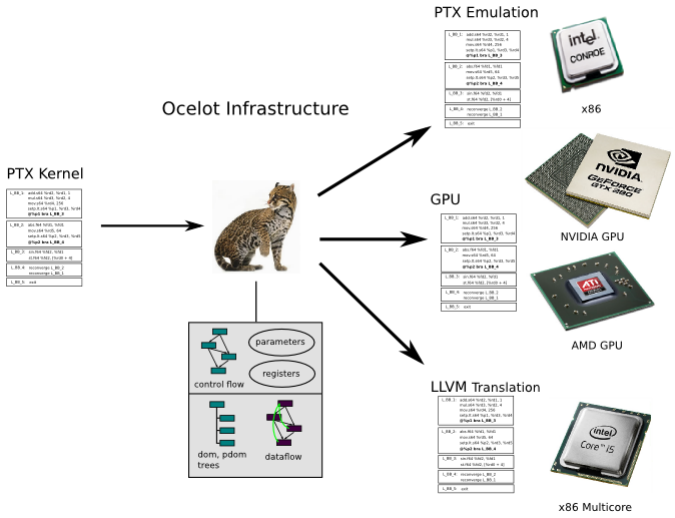


Figura 3.2: Input e output dell'applicazione GPUoceto per la conversione di kernel PTX

DirectX né sforzi per far apparire normali problemi come problemi di tipo grafico. A partire da 2008 NVIDIA ha aggiunto al suo SDK anche la possibilità di utilizzare OpenCL al posto di CUDA C fornendo un ulteriore modo di sfruttare i suoi prodotti per il GPU computing. Per sfruttare il codice CUDA C l'SDK fornito da Nvidia mette a disposizione il compilatore *nvcc* che produce codice PTX, un codice pseudo-assembly che durante l'esecuzione è tradotto dai driver Nvidia in codice binario eseguibile dalla GPU. Anche il codice dei kernel OpenCL è compilato in PTX per essere utilizzato, per OpenCL si passa prima da una rappresentazione intermedia LLVM IR (Low Level Virtual Machine Intermediate Representation).

L'implementazione OpenCL di Nvidia non supporta l'esecuzione su CPU né tantomeno CUDA C (che può essere compilato solo in codice PTX) può essere utilizzato su CPU, tuttavia ci sono sforzi da parte di Nvidia e altre organizzazioni per utilizzare codice CUDA anche su CPU. The Portland Group (supportata da Nvidia) fornisce il compilatore "PGI CUDA C for x86" (`http://www.pgroup.com/resources/cuda-x86.htm`), si fa notare che questo progetto riguarda la compilazione, il codice CUDA C deve essere quindi ricompilato e il progetto non fornisce alcun modo di utilizzare applicazioni CUDA su hardware che non includono GPU adatte. Il progetto GPUocelot fornisce invece un modo per utilizzare kernel già compilati in codice PTX sulle GPU di AMD oppure su CPU utilizzando tecniche quali emulazione e traduzione LLVM.

3.3.3.3 Applicazioni

Dopo il suo debutto nel 2007 varie industrie e applicazioni hanno utilizzato CUDA C. Si elencano qui di seguito alcuni ambiti in cui CUDA è stato impiegato con successo:

- Immagini mediche: la TechnicScan Medical Systems ha sviluppato un sistema per ricavare immagini 3D da ultrasuoni in modo da ridurre i rischi dovuti all'esposizione ripetuta ai raggi X nel caso di mammografia e questo sistema veniva utilizzato in alcuni casi in congiunzione alle normali mammografie ma richiedeva un tempo di calcolo eccessivamente alto per un uso pratico costante. Dopo l'introduzione di CUDA la Techniscan ha realizzato il sistema Svara ad ultrasuoni. Questo sistema utilizza due schede Nvidia Tesla C1060.

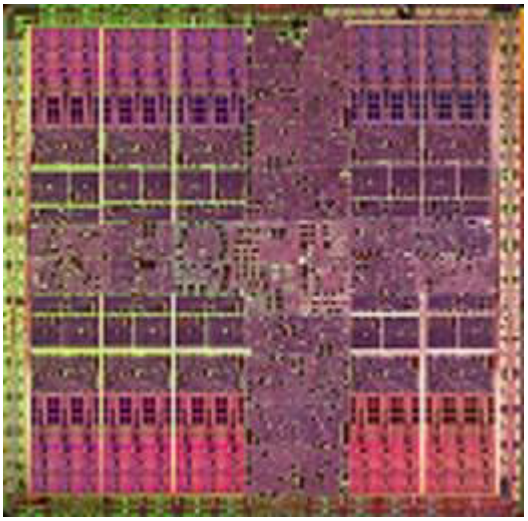


Figura 3.3: La GPU Tesla C1060

- Fluidodinamica computazionale: per molti anni la progettazione di pale e rotori era rimasta un'arte oscura poiché solo i supercomputer potevano simulare il movimento dell'aria intorno alle pale con modelli accurati ma il Dr. Graham Pullan e il Dr. Tobias Brandvik del "many-core group" di Cambridge hanno identifiato in CUDA il potenziale per accelerare i calcoli fluidodinamici. All'inizio hanno investigato single workstation con GPU e poi cluster di GPU riuscendo ad ottenere alte prestazioni a basso costo.
- Scienze ambientali: negli ultimi anni è cresciuto sempre più l'interesse ad offire prodotti rispettosi dell'ambiente, ad esempio molti ricercatori hanno cercato di diminuire l'impatto ambientale dei detergenti senza ridurne l'efficacia. L'efficacia dei detergenti è dovuta alla presenza di tensioattivi

(o surfattanti) che hanno però effetti dannosi sull'ambiente. Queste sostanze si legano allo sporco in modo che poi possano essere portate via con l'acqua insieme allo sporco. Tradizionalmente si usano molti esperimenti di laboratorio utilizzando diverse combinazioni di materiali e impurità da eliminare. La Temple University ha lavorato con la Procter & Gamble, uno dei leader del settore, per usare simulazioni molecolari delle interazioni dei tensioattivi con con lo sporco, l'acqua e altri materiali. Queste simulazioni permettono di testare molte più combinazioni di quelle eseguibili in laboratorio. I ricercatori della Temple hanno usato il software di simulazione HOOMD (Highly Optimized Object-oriented Many-particle Dynamics) realizzato dall'Ames Laboratory del Dipartimento dell'Energia americano. Distribuendo le loro simulazioni su due GPU Tesla di NVIDIA hanno ottenuto prestazioni equivalenti a quelle del Cray XT3 e dell'IBM BlueGene/L. Aumentando il numero di GPU Tesla hanno migliorato le performance di 16 volte. Questo approccio probabilmente migliorerà l'efficacia dei prodotti diminuendo l'impatto ambientale.

3.3.4 ATI

La storia del GPU computing ATI (e poi AMD, la quale ha terminato l'acquisizione di ATI nel 2006) riportata qui di seguito è tratta da [AMD, 2010b].

Nel 2006 AMD ha introdotto la tecnologia "Close to Metal" (CTU) per il GPGPU, questo standard permette agli sviluppatori l'accesso all'insieme nativo di istruzioni e alla memoria della GPU di AMD. Usando questa tecnologia le GPU sono diventate di fatto potenti architetture aperte programmabili come le CPU. CTM forniva agli sviluppatori accesso all'hardware a basso livello, ripetibile e deterministico. Quest'accesso era utilizzabile per sviluppare compilatori, debuggers, librerie matematiche e piattaforme per le applicazioni.

CTM è stato utilizzato per produrre un client per Folding@home trenta volte più veloce del corrispettivo per CPU.

Collaborando con lo sviluppatore di strumenti RapidMind Inc. AMD ha sviluppato uno stack di sviluppo multistrato per il software GPGPU, questo permetteva agli sviluppatori di utilizzare un linguaggio di alto livello permettendo comunque agli utenti più esperti di controllare direttamente tutte le risorse hardware. Nello stesso periodo AMD ha presentato la prima generazione di FireStream come primo hardware di stream processing (elaborazione di flussi di dati) disponibile commercialmente.

3.3.4.1 ATI Stream

Nel dicembre 2007 AMD fornì un ambiente di sviluppo che esponeva la capacità computazionale delle proprie GPU: l'ATI Stream SDK.

L' ATI Stream SDK fornì agli sviluppatori la possibilità di utilizzare ATI Brook+, un linguaggio ad alto livello. CTM venne evoluto in ATI CAL (Compute Abstaction Layer), lo strato API di supporto per Brook+. L'introduzione di Brook+ significò che AMD poteva fornire un'intero stack di sviluppo, dal

livello più alto a quello più basso. Nel 2007 AMD introdusse la seconda generazione di soluzioni per lo stream processing, l'AMD FireStream 9170. La 9170 rivoluzionò il GPGPU fornendo per la prima volta il supporto hardware ai calcoli in virgola mobile a doppia precisone.

Nel giugno del 2008 AMD partecipò alla formazione del gruppo di lavoro per la definizione di OpenCL sotto il Khronos Group e successivamente (vedi la sezione corrispondente) introdusse la terza generazione di soluzioni per lo stream processing superando la barriera del TFLOPS in in un singolo slot PCI-Express (secondo quanto riportato AMD con migliori prestazioni e consumo ridotto rispetto alla concorrenza).

Nel dicembre del 2008 venne pubblicata la prima specifica di OpenCL e AMD integrò le librerie runtime di CAL nei driver Catalyst sbloccando le capacità ATI Stream di molte schede grafiche già presenti sul mercato. Nel 2009 AMD ha integrato OpenCL nell'ATI Stream SDK e nel terzo quadrimestre dello stesso anno ha rilasciato la FireStream 9270 che fornisce 1.2 TFLOPS assorbendo 160W di potenza elettrica.

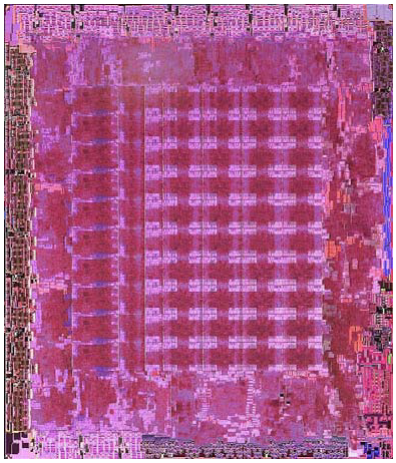


Figura 3.4: ATI RV770 corrispondente alle GPU Radeon HD: 4870 (RV770XT), 4850 (RV770PRO) e 4830 (RV770PROLE)

L'ATI Stream SDK ha assunto poi nel 2011 la denominazione di AMD APP (Accelerated Parallel Processing) SDK.

L'SDK utilizza per la compilazione dei kernel diversi livelli di compilazione, in particolare poiché uno stesso codice deve poter essere compilato sia per CPU sia per GPU.

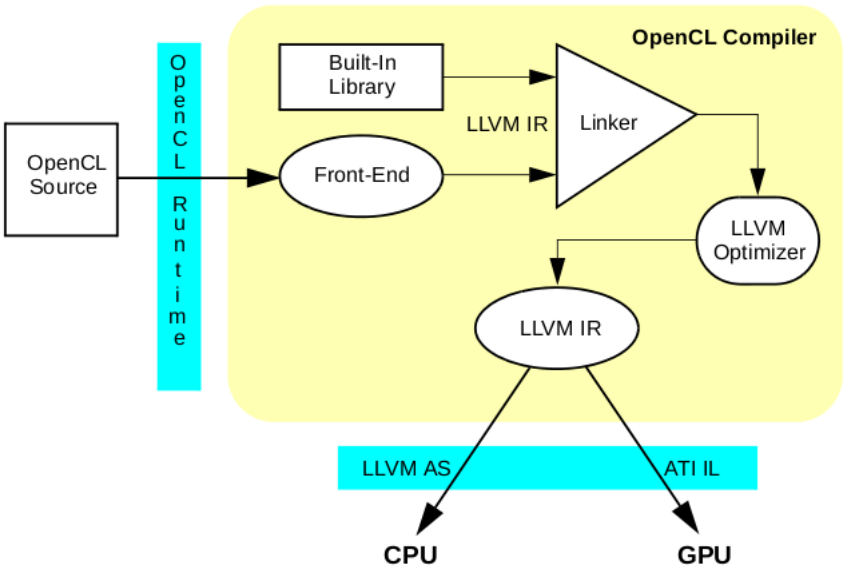


Figura 3.5: La catena di compilazione OpenCL nell'AMD APP SDK (e nel precedente ATI Stream SDK)

Il front-end include diverse trasformazioni di alto livello che trasformano il codice del kernel in codice LLVM IR, il back-end utilizza ottimizzazioni diverse in base al tipo di dispositivo.

Per la CPU il runtime OpenCL utilizza LLVM AS (LLVM assembler) per trasformare la rappresentazione intermedia in codice x86 (oppure x86_64), il runtime OpenCL è in grado di determinare automaticamente il numero di core disponibili. Per quanto riguarda la GPU il runtime OpenCL effettua post-processing sul codice ATI IL (ATI Intermediate Language) prodotto dal compilatore OpenCL aggiungendo macro specifiche della GPU dopodiché il runtime OpenCL rimuove le funzioni non necessarie e passa il codice IL completo al compilatore CAL che è in grado di produrre file binari specifici eseguibili dalla GPU.

3.3.5 Presente e futuro del calcolo eterogeneo

I supercomputer attuali sfruttano spesso ambienti eterogenei che utilizzano sia molte CPU sia molte GPU (o altri processori specializzati) in modo da sfruttare entrambe le architetture per i calcoli su cui ognuna è più efficiente, questa direzione sarà percorsa anche in futuro.

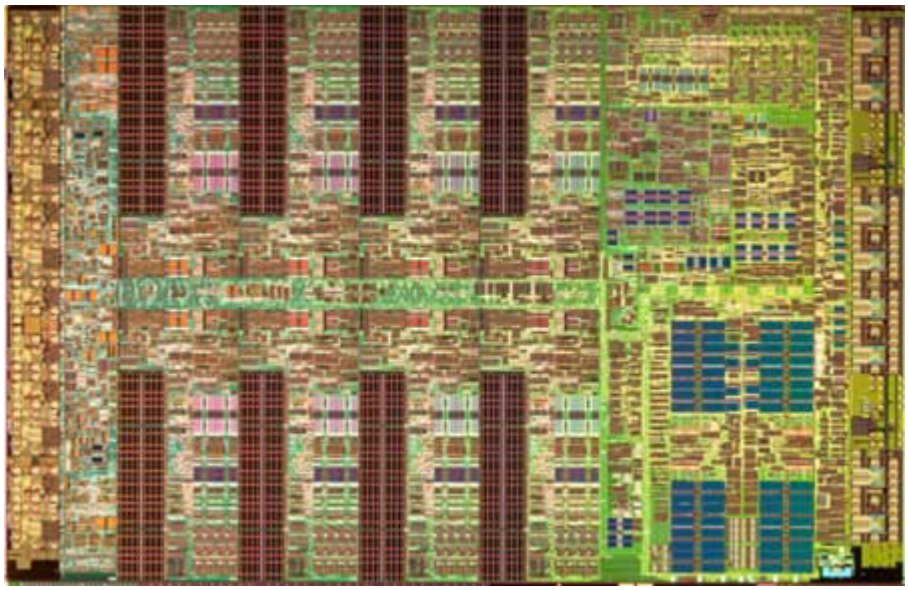


Figura 3.6: Il microprocessore IBM Cell Broadband Engine

Come esempi si possono portare:

- il supercomputer IBM Roadrunner (che supera il petaflop) è realizzato con CPU AMD Opteron e processori Cell Broadband Engine, questi ultimi non sono GPU ma tale supercomputer è comunque un buon esempio di ambiente eterogeneo
- il sistema Mole-8.5 è realizzato con processori Xeon e NVIDIA Tesla C2050 (`http://i.top500.org/system/176899`) ed è in grado di simulare in un giorno 770 picosecondi del comportamento completo del virus H1N1 (sottotipo del virus dell'influenza A) sviluppando una potenza computazionale di 1 petaflop in doppia precisione e 2 petaflop in singola (`www.twiv.tv/11yt2114.pdf`)
- il progetto europeo Mont-Blanc realizzerà un supercomputer basato su moduli con processore ARM basato su Tegra e GPU di tipo GeForce (`http://www.montblanc-project.eu/`). Il software utilizzerà la libreria OpenACC per indicare quali parti di codice (C, C++ o Fortran) possono essere accelerate su dispositivi dedicati (in questo caso sarà prodotto codice per le GPU)
- Cray prevede di costruire un supercomputer denominato Titan con 18000 CPU Opteron e altrettante GPU Nvidia
- Nvidia sta preparando la costruzione di un supercomputer di potenza superiore a un exaflop con il progetto "ExaScale Machine"

Anche nel mercato per l'utilizzo domestico si dimostra sempre più attenzione nel costruire piattaforme eterogenee, Intel produce molti processori con grafica integrata, AMD fornisce la piattaforma Fusion che unisce CPU e GPU su un singolo chip e gli smartphone utilizzano system-on-a-chip comprensivi di CPU e GPU.

3.4 OpenCL

OpenCL (Open Computing Language) è uno standard di programmazione aperto (non proprietario e indipendente dal produttore di hardware) e libero da royalty per calcoli di tipo general-purpose su sistemi eterogenei. Questo standard è stato inizialmente propposto da Apple Inc. e poi sviluppato dal Khronos Group per permettere ai programmatori di scrivere codici che abbiano come piattaforma di esecuzione sia le CPU multi-core sia le GPU più moderne.

OpenCL è uno standard a più livelli infatti le sue specifiche riguardano tre diversi aspetti: il linguaggio, le API del platform layer e le API di runtime.

Il primo aspetto descrive la sintassi e l'interfaccia di programmazione per scrivere kernel (saranno descritti in seguito) che possano essere eseguiti sulle CPU e GPU supportate. Il linguaggio è basato su un sottoinsieme dello standard ISO C99. Il linguaggio C è stato scelto come base per il primo linguaggio di scrittura dei kernel a causa della sua diffusione tra gli sviluppatori e della familiarità che ne hanno. Per garantire coerenza di risultati tra le diverse piattaforme la ben definita accuratezza numerica IEEE 754 è stata definita per tutte le operazioni in virgola mobile con un ricco insieme di funzioni built-in. Interessante la possibilità dello sviluppatore di pre-compilare i suoi kernel per uno specifico ambiente oppure di lasciare che sia l'ambiente di esecuzione (runtime) a compilarli su richiesta (on demand).

Le API di piattaforma (platform layer) danno allo sviluppatore accesso a routine che possono richiedere il numero e il tipo dei dispositivi nel sistema. Lo sviluppatore può selezionare, sulla base delle risposte, e inizializzare i dispositivi che possano eseguire adeguatamente il carico di lavoro, a questo livello vengono creati i contesti di calcolo, le code per l'inserimento delle attività (jobs) e le richieste per il trasferimento di dati (non si tratta di attività automatiche, lo sviluppatore deve scegliere alcuni parametri sulla base delle sue necessità).

Le API di runtime permettono allo sviluppatore di inserire nelle code create i kernel per la loro effettiva esecuzione e sono responsabili della gestione delle risorse di calcolo e di memoria.

Il seguente elenco tratto dalla tabella disponibile all'indirizzo [AMD, 2010a] riassume i diversi aspetti della specifica di OpenCL.

- La specifica di linguaggio descrive
  - un'interfaccia di programmazione multipiattaforma basata sul C
  - un sottoinsieme del linguaggio ISO C99 (familiare agli sviluppatori) con estensioni di linguaggio

  - accuratezza numerica ben definita (approssimazione secondo lo standard IEEE 754 con massimo errore definito)
  - la possibiltà di compilare in linea (runtime) oppure offline (a tempo di compilazione) e il building dei kernel di computazione come eseguibili
  - un ricco insieme di funzioni già implementate (built-in)

- La specifica del Platform Layer API descrive
  - uno strato di astrazione hardware (hardware abstraction layer) su diverse risorse computazionali
  - la possiblità di interrogare (query), selezionare (select) e inizializzare i dispositivi di calcolo (compute devices)
  - la creazione di contesti di calcolo (compute contexts) e code di esecuzioni (work-queues)

- La specifica delle API di runtime descrive
  - l'esecuzione dei kernel di computazione
  - la gestione della schedulazione, della computazione e delle risorse di memoria

Poiché OpenCL è uno standard deve avere un modello di esecuzione indipendente dalla piattaforma che possa essere eseguito da diversi dispositivi (anche se per comodità è molto vicino a quello delle GPU). I kernel di OpenCL possone essere pensati come base sia per esecuzioni parallele sui dati (che ben si adatta alle GPU o a processori con istruzioni vettoriali come SSE cioè Streaming SIMD Extensions) sia per esecuzioni parallele di attività (come quelle eseguite normalmente sulle CPU multi-core che possono distribuire processi o thread completamente diversi sui diversi core).

Un kernel di computazione è l'unità base di codice eseguibile (cioè piccoli programmi sviluppati dall'utente che sono eseguiti ripetutamente su un flusso di dati) e può essere assimilato ad una funzione C (viene infatti implementato in tale modo).

Come descritto nella sezione relativa all'evoluzione del GPU computing i primi esperimenti utilizzavano pixel shader per effettuare calcoli generici infatti le schede video permettono l'esecuzione di kernel di diverso tipo: vertex, pixel, geometry, domain, hull e, nelle GPU più recenti, compute [AMD, 2011a] (ovviamente supportato da tutte le GPU conformi a OpenCL). Nel seguito si indicheranno i compute kernel (o kernel di computazione) solo come kernel.

L'esecuzione dei kernel implementati può avvenire sia in-order sia out-of-order sulla base dei parametri scelti al momento dell'inserimento in coda. L'esecuzione in-order garantisce l'esecuzione delle code nell'ordine di inserimento e l'esecuzione di un comando non viene avviata fino a che non è terminata quella del comando precedente (ed è l'unica supportata dal runtime AMD), quella out-of-order non fornisce garanzie né sull'ordine di esecuzione né su quanti comandi

possono essere in esecuzione in un dato momento. Sono disponibili eventi che permettono allo sviluppatore di controllare lo stato delle richieste di esecuzione dei kernel e delle altre richieste runtime, tali eventi possono anche essere utilizzati per effettuare delle sincronizzazioni.

In termini di organizzazione il dominio di esecuzione di un kernel è definito come un dominio di computazione N-dimensionale (nella specifica attuale si possono usare da 1 a 3 dimensioni). Questo permette al sistema di conoscere quanto sia grande il problema a cui l'utente vuole che i kernel siano applicati. Per essere più specifici il programmatore deve decidere il numero di istanze di un kernel che sono chiamate work-item e il modo di identificarli che può essere composto da una o più coordinate (fino a 3), questo si applica naturalmente alle applicazioni in cui ogni work-item è associato a un calcolo che dovrà riempire un elemento di un vettore o di una matrice a 2 o 3 dimensioni (non è comunque obbligatorio utilizzare uno work-item per ogni cella da riempire). Per permettere la comunicazione tra diversi work-item e poterli sincronizzare è possibile dividerli in work-group, in questo modo la comunicazione e la sincronizzazione tra work-item sono effettuate solo all'interno di un gruppo e i diversi group possono essere eseguiti in modo indipendente l'uno dall'altro. È sempre possibile utilizzare la memoria globale (spiegata tra breve) per la comunicazione tra diversi work-group (facendo recuperare ad altri work-group risultati inseriti scritti nella memoria globale) ed è disponibile un modo per creare dei punti di attesa globale anche all'interno del kernel (se ad esempio diversi item hanno bisogno sia di scrivere sia di leggere valori calcolati da altri item nella stessa coda di esecuzione) tuttavia questi approcci sono da evitare se non indispensabili in quanto riducono notevolmente o vanificano l'utilità di suddividere gli work-item in work-group. L'indipendenza tra work-group ha il pregio di rendere le applicazioni OpenCL molto scalabili infatti è possibile aumentare o diminuire le dimensioni dell'intera esecuzione o del singolo work-group senza doversi preoccupare degli effetti sulla comunicazione tra questi. La divisione in work-group è necessaria in caso si voglia condividere la memoria tra diversi item (questa memoria condivisa è denominata memoria locale nello standard OpenCL) tuttavia è comunque possibile indicare una dimensione di uno work-group per dividere gli work-item in gruppi distribuibili sulle diverse unità di calcolo senza utilizzare la memoria condivisa. La sincronizzazione globale è invece necessaria ogni volta che si vogliono recuperare nella memoria dell'host, cioè la macchina su cui è montato il dispositivo OpenCL, i dati calcolati dagli item e inseriti in memoria globale, l'host deve attendere l'esecuzione di tutti gli item altrimenti potrebbe trovare valori indefiniti nel caso non tutti gli item abbiano finito l'esecuzione (questa sincronizzazione non avviene però all'interno del kernel, è definita nel codice host come attesa della terminazione della coda di esecuzione). La sincronizzazione è anche possibile tra i diversi comandi di una *command queue* in un singolo contesto, cioè è possibile utilizzare kernel diversi per avviare diverse esecuzioni nella stessa *command queue* ed è possibile sincronizzare le diverse esecuzioni tramite eventi (può essere il caso ad esempio di una ricerca parallela di minimo che prima calcoli dei minimi locali e poi scelga il minimo tra questi).

Anche la memoria in OpenCL è descritta nello standard e deve essere divisa

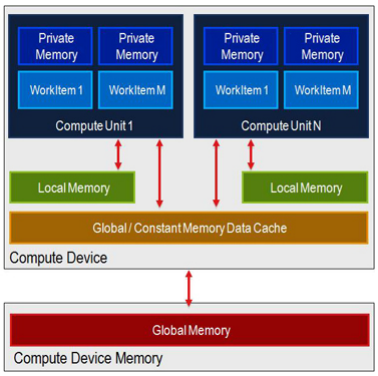


Figura 3.7: Struttura della memoria secondo lo standard OpenCL

in zone dalla diversa visibilità, da quella privata che è la più restrittiva a cui si può accedere solo dalla corrispondente unità di calcolo fisica (a cui di solito è associato un singolo work-item) a quella globale il cui accesso può essere effettuato da tutte le unità di computazione, a seconda dell'effettivo sottoinsieme di memoria diversi tipi di memoria possono essere fusi insieme.

OpenCL 1.0 definisce 4 spazi di memoria: privato, locale, costante e globale.

La memoria privata (private) è la memoria che può essere usata da una singola unità di computazione come i registri di una unità di computazione (computation unit) in una GPU oppure come un singolo core di una GPU, è usata automaticamente dalla variabili interne di un kernel.

La memoria locale (local) può essere usata da diversi work-item che sono nello stesso work-group, quindi condivisa tra questi, un esempio di come possa essere usata è quello di costruire un vettore, nel caso di ricerca di massimo o di minimo, in modo che ogni work-group possa mettere il suo risultato parziale in una cella (il vettore può essere acceduto da tutti ma ogni cella è associata ad un solo work-item e vice versa e ogni cella "risultato" è associata a un solo work-group). Questo è simile al local data share che è disponibile nelle moderne GPU prodotte da AMD e alla memoria shared di CUDA. Per essere utilizzata la memoria locale deve essere indicata esplicitamente.

La memoria costante (constant) può essere usata per memorizzare dati costanti per un accesso in sola lettura da parte di tutte le unità di computazione presenti nel dispositivo e utilizzate durante l'esecuzione. Il processore ospite (host) è responsabile dell'allocazione e dell'inizializzazione degli oggetti di memoria che risiedono in questo spazio di memoria; questo è simile alle cache costanti presenti nelle GPU di AMD.

Infine la memoria globale (global) è la memoria che può essere usata da tutte le unità di computazione presenti sul dispositivo, questo è simile alla memoria off-chip di una GPU (in pratica della memoria che si trova sulla scheda video

e non internamente alla GPU). L'uso di questo tipo di memoria è necessario per il passaggio dei parametri agli work-item e per la restituzione dei risultati che altrimenti rimarrebbero intrappolati nella GPU, è infatti l'unico modo in cui un dispositivo OpenCL come una GPU può comunicare con l'esterno e tali trasferimenti di memoria tra *host* e *device* sono espliciti.

Bisogna anche aggiungere che nel caso si voglia utilizzare OpenCL per eseguire i calcoli su una GPU sono presenti altri tipi di memoria all'esterno del dispositivo utilizzato, poiché essa deve comunicare con un sistema ospite, che sono la memoria PCIexpress (che è parte della memoria RAM accessibile sia alla CPU sia alla GPU) e la memoria host (dell'ospite, la normale memoria RAM del computer acceduta dai programmi), si precisa quindi che la memoria globale non risiede nella normale RAM ma è comunque legata al dispositivo e sono necessari dei trasferimenti tra la memoria RAM e la memoria globale per poter passare dei parametri agli work-item e i trasferimenti inversi per restituire i risultati. Il passaggio dei dati è costituito da due fasi di copia: dalla memoria dell'ospite alla PCIexpress e da quest'ultima alla memoria globale e può essere richiamata implicitamente oppure esplicitamente nel caso si ritenga di usare uno solo dei due versi per la copia (creando un buffer sul dispositivo in sola lettura o in sola scrittura). Esistono delle tecniche con specifiche restrizioni come i limiti di pagina e l'allineamento di memoria per evitare l'uso della la copia tra la memoria normale e la PCIexpress rimappando la memoria host direttamente all'interno della memoria PCIexpress, tuttavia è un approccio difficile e propenso all'errore.

Qui di seguito si riporta lo schema [AMD, 2011a] delle interazioni tra i diversi tipi di memoria nell'architettura ATI Stream

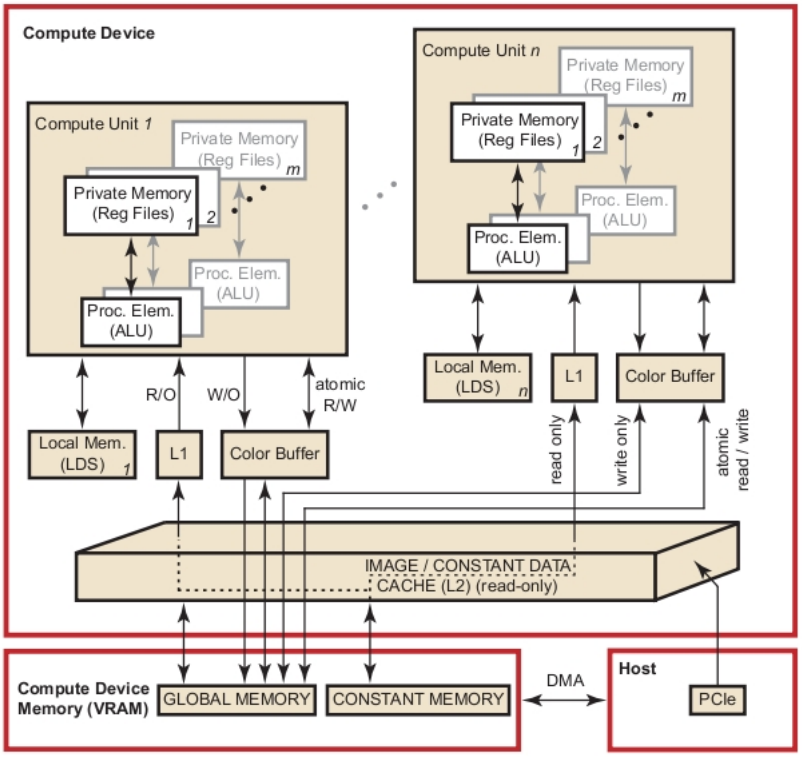


Figura 3.8: Interazioni tra i diversi tipi di memoria nell'architettura ATI Stream

e qui di seguito il flusso standard dei dati [AMD, 2011a] tra host (CPU) e GPU, si noti che anche possibile passare direttamente dalla memoria globale a quella privata senza passare dalla memoria locale (come indicano le frecce grigie tratteggiate).

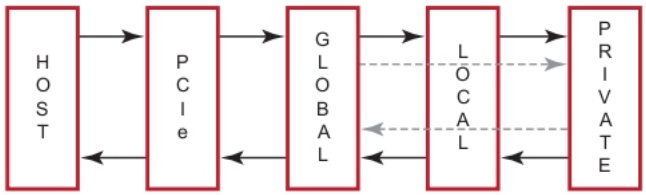


Figura 3.9: Flusso standard tra host e GPU dei dati in OpenCL

I *device* di OpenCL corrispondono normalmente ai dispositivi fisici, è comunque possibile utilizzare una tecnica denominata *device fission* per utilizzare un solo dispositivo reale (come una GPU) come fossero più di uno, ad esempio per riutilizzare tecniche di sincronizzazione già note.

Per un esempio dei possibili usi di OpenCL ci si può riferire alle simulazioni dell'attrazione gravitazionale tra n corpi [BDT, 2011]. Anche NVIDIA insieme alle altre applicazioni OpenCL di esempio per le proprie GPU ne propone una simile `http://developer.download.nvidia.com/compute/cuda/3_0/sdk/website/OpenCL/website/samples.html` funzionante su GeForce della serie 8 e GPU successive.

3.4.1 Altre aziende che supportano OpenCL

Anche nel settore mobile e embedded c'è interesse per OpenCL: Imagination ha annunciato l'intellectual property core della GPU POWERVR SGX545 per i settori mobile ed embedded dichiarato compatibile con OpenCL 1.0 (oltre che con OpenGL 3.2 e DirectX 10.1) ma non fornisce un SDK correlato, ARM ha annunciato la disponibilità della GPU mobile Mali-T604 compatibile con OpenCL. Per il settore mobile anche Nvidia supporta OpenCL con la piattaforma Ion (la piattaforma Tegra per ora non lo supporta ma è previsto il supporto a partire dalla serie "Wayne" della piattaforma Tegra 3).

Anche altre aziende produttrici di hardware oltre a quelle di GPU (AMD e NVIDIA) supportano OpenCL per permettere ai programmatori di sfruttare in maniera semplice il parallelismo insito nei loro prodotti (si ricorda che OpenCL è uno standard per ambienti eterogenei, non solo per il GPU computing). Intel fornisce un SDK per OpenCL (per Windows nelle versioni a 32 e a 64 bit e per GNU/Linux nella sola versione a 64 bit) compatibile con i suoi processori multicore marchiati Core e Xeon (per ora non è previsto un supporto per le GPU integrate). IBM fornisce un SDK per OpenCL per GNU/Linux su architettura Power da utilizzare sui prodotti BladeCenter (dotati eventualmente del processore Cell Broadband Engine).

Infine Apple Inc. fornisce una propria implementazione di OpenCL inclusa nel proprio sistema operativo a partire da Mac OS X 10.6 "Snow Leopard".

3.5 Tipi di parallelismo gestibili in OpenCL

Questa sezione e la seguente sono basate principalmente sulle diapositive dell'AMD OpenCL University Kit e da esso sono anche tratte le immagini riportate. La parallelizzazione può essere automatica a livello di istruzione (instruction level parallelism, particolarmente sviluppato durante gli anni '90 per migliorare le prestazioni delle CPU) oppure a più alto livello. Per il parallelismo a livello di istruzione i processori usano diverse tecniche, come ad esempio l'utilizzo di più *pipeline* per eseguire contemporaneamente istruzioni indipendenti. OpenCL si pone invece al livello più alto, la parallelizzazione non è automatica ma è necessario istruire il software, decidere cioè durante la scrittura del programma quali

parti parallelizzare, un modello di programmazione (cioè in che modo farlo) e l'hardware più adatto su cui eseguire il programma così costruito.

Esistono principalmente due modi di decomporre un algoritmo in calcoli paralleli:

- per compito (*task-driven*): in questo caso ogni thread o unità hardware esegue una parte dell'algortimo (un compito) in parallelo
- sui dati (*datadriven*): in questo caso ogni thread o unità hardware esegue la stessa attività ma su dati diversi

L'approccio più naturale per sfruttare OpenCL è la decomposizione sui dati utilizzando una corrispondenza uno a uno tra i dati in memoria e gli work-item basandosi sul loro indice (è possibile in questo caso definire gli work-group esplicitamente o implicitamente al contrario di CUDA che richiede sempre di indicare le dimensione di un blocco). La decomposizione sui dati può essere effettuata sul numero di dati in ingresso (quando si basa su funzioni uno a molti) o in uscita (quando si basa su funzioni uno a uno o molti a uno). La decomposizione sui dati in uscita è quella più diffusa in ambito scientifico e ingegneristico come nel caso dei pixel di un'immagine ottenuta per convoluzione (si veda a proposito l'esempio AMD citato in 4.6) o una matrice ottenuta per moltiplicazione (il numero di celle in output è inferiore alla somma del numero di celle delle matrici utilizzare nella moltiplicazione). Si può anche eseguire una parallelizzazione per attività in modo indipendente dagli indici degli work-item accodando kernel diversi nella stessa coda o avviando più *command queue* contemporaneamente (eventualmente per utilizzare tipi di vettori specifici per ogni dispositivo). Se le *command queue* sono nello stesso contesto è anche possibile sincronizzarle tramite eventi.

Come è noto, esistono quattro tipi di parallelizzazione secondo la tassonomia di Flynn.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Tabella 3.1: Tassonomia di Flynn per le architetture parallele

- SISD: l'architettura più semplice, un'istruzione alla volta utilizza un dato alla volta senza alcuna parallelizzazione
- SIMD: una singola istruzione è in grado di essere eseguita contemporaneamente su più dati, è il caso delle istruzioni SSE dei processori o dell'architettura interna delle GPU. Anche i supercomputer possono utilizzare questa architettura, come ad esempio la famiglia italiana APE100. Questo approccio riduce la necessità di controllo di flusso ma aumentà la necessità

di ALU (per eseguire contemporaneamente quattro somme, ad esempio, sono necessarie 4 ALU ma non è necessario gestire un ciclo per scandire un vettore).

- MISD: molte istruzioni possono essere eseguite sugli stessi dati, questo può essere sfruttato ad esempio per la replicazione del programma in modo che anche nel caso si rompa l'hardware responsabile di una delle esecuzioni il programma possa comunque continuare a essere eseguito senza interruzioni
- MIMD: diverse istruzioni operano su dati diversi, è il caso dei computer distribuiti

L'utilizzo di OpenCL ricade normalmente nel caso SPMD (cioè Single Program Multiple Data, in particolare per l'esecuzione GPU) per l'esecuzione generale del programma infatti i diversi work-item eseguono lo stesso programma (il kernel) su dati diversi. Questo caso non è riportato nella tabella precedente perché si tratta di un'esecuzione parallela a livello di programma e non di istruzione, gli work-item non hanno in comune una singola istruzione ma un intero programma, si può comunque considerare un caso particolare del caso MIMD infatti anche se le istruzioni sono le stesse poiché appartenenti allo stesso programma non è garantito che diversi work-item o diverse unità hardware stiano eseguendo la stessa istruzione in un dato momento quindi l'esecuzione consiste in istruzioni diverse che sono eseguite contemporaneamente su dati diversi. OpenCL supporta operazioni atomiche, cioè operazioni di lettura o scrittura effettuate senza l'intervento di altri item, alcune GPU supportano operazioni atomiche a livello di sitema introducendo quindi una sincronizzazione globale riducendo le performance, algoritmi che richiedano necessariamente queste operazioni non sono adatti alle GPU, come anche gli algoritmi che richiedono una decomposizione sull'input, devono perciò essere ristruttuati per un'esecuzione efficiente su GPU. Altri modi, oltre al GPU computing, che possono utilizzare un approccio SPMD sono MPI (Message Passage Interface) su un cluster distribuito e i thread POSIX (implementati in UNIX e Linux) in un sistema con memoria condivisa. Utilizzando una decomposizione sulle attività invece si può descrivere l'esecuzione con MPMD (Multiple Program Multiple Data) si eseguono contemporaneamente attività diverse (con diversi programmi) su dati diversi. Utilizzando un'approccio di questo tipo su GPU le *command queue* associate alle diverse attività non possono comunicare. Il kernel può al suo interno utilizzare tipi di dato composti da più elementi (ad esempio *float4*) utilizzando così un approccio SIMD (in questo modo si possono sfruttare le istruzioni SSE dei processori o aumentare le prestazioni sfruttando l'architettura specifica delle GPU). Si precisa che l'hardware della GPU applica automaticamente gli work-item di tipo SPMD sulle unità di tipo SIMD interne (questo viene a volte definito SIMT, Single Instruction Multiple Thread) quindi esistono esecuzioni SIMD ma l'esecuzione complessiva rimane SPMD, una gestione esplicita può essere considerata solo per migliorare le prestazioni.

Una delle tecniche della parallelizzazione è l'eliminazione dei cicli in esecuzioni parallele ed è chiamata *loop strip mining* e può essere effettuata in diversi

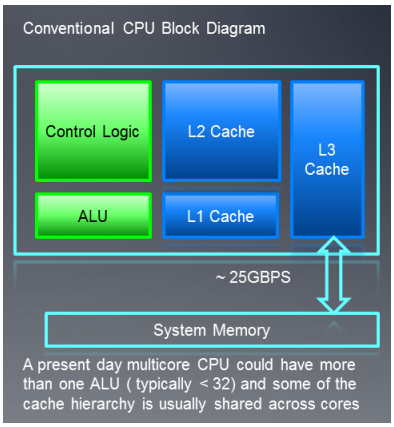


Figura 3.10: Architettura convenzionale di una CPU

modi: con un'approccio SIMD eseguendo operazioni su più dati con la stessa istruzione (per una somma di due vettori ad esempio utilizzare le istruzioni SSE per effettuare la somma su quattro celle alla volta) oppure usando diversi thread con un metodo SPMD. Il secondo modo può essere a sua volta distinto in una tecnica "a sezioni" e in una tecnica massicciamente parallela. La prima utilizza un numero definito di thread ed esegue le operazioni su una sezione del vettore, questo è adatto alle CPU che hanno poche unità parallele (core) e hanno un maggiore *overhead* per la creazione di thread ma possono gestire meglio controlli di flusso più complessi come i cicli, il secondo invece adatto più alle GPU consiste nel creare un thread (work-item in OpenCL) per ogni dato (cella del vettore) in output.

In conclusione l'approccio migliore per l'esecuzione OpenCL su GPU è SPMD (sugli work-item) con decomposizione *massiccia* sui dati (in particolare di output per applicazioni scientifiche) ed esecuzione SIMD in ogni item. Per le CPU invece l'approccio migliore è un numero ridotto di thread utilizzando MIMD con decomposizione sulle attività o SPMD con decomposizione sui dati, utilizzando sempre un numero ridotto di thread.

3.6 Differenze essenziali tra CPU e GPU

Come già accennato le CPU e le GPU sono adatte ognuna a diversi tipi di parallelizzazione, questo deriva dalla loro architettura interna derivante dagli scopi per cui sono state costruite.

Le CPU sono composte da:

- ALU (Arithmetic Logic Unit): responsabile delle operazioni aritmetiche e logiche. Una CPU moderna ne contiene molte, tipicamente un umero superiore a 32

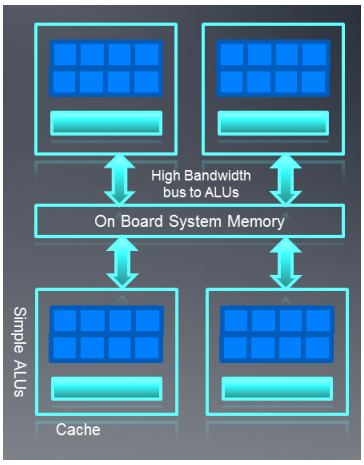


Figura 3.11: Architettura di una moderna GPU

- Control logic: responsabile di gestire al meglio il controllo di flusso riordinando l'esecuzione, fornendo parallelismo a livello di istruzione (ILP) e minimizzando i *pipeline stalls*
- Cache multilivello: è utilizzata per nascondere la latenza evitando l'accesso continuo alla memoria. Attualmente le CPU hanno fino a tre livelli di cache e alcuni di essi possono essere divisi fra più core
- Registri: un numero limitato a causa del numero ridotto di thread attivi gestibili

Le GPU invece sono composte da:

- molti più "core" generici rispetto alle CPU costituiti da:
  - molte ALU ognuno (un numero molto superiore a quello delle CPU considerando il totale della GPU)
  - grandi file di registro per supportare l'esecuzione di molti thread in un solo contesto
  - meno spazio dedicato alla logica di controllo e alla cache rispetto alle CPU
- hardware a bassa latenza per gestire lo *switch* di thread da eseguire
- un bus di memoria con ampia larghezza di banda che riesce a servire molte ALU contemporaneamente, la tecnica per generare codice efficiente per una GPU è quindi quella di sfruttare al meglio l'ampiezza di banda disponibile

Qui di seguito si riportano maggiori dettagli relativi all'architettura di recenti GPU prodotte da AMD ed Nvidia, tutta questa struttura è comunque trasparente dal punto di vista dell'utilizzo di OpenCL e il programmatore può soltanto conoscere il numero di *compute units* tramite un'apposita funzione e alcuni limiti di memoria (e di numero di work-item gestibili), le informazioni possono essere utili nel caso si disponga di un'unico dispositivo e il programmatore voglia strutturare il programma in modo da sfruttare al massimo la potenza computazionale del singolo dispositivo per ricercare specifici risultati e non sia interessato a realizzare un'applicazione distribuibile su hardware diverso (è il caso di supercomputer costruiti ad hoc come nell'esempio per la ricerca dei tensioattivi con schede Tesla riportato nella sezione 3.3.3.3 rigurdante le applicazioni di CUDA).

3.6.1 Architettura Cypress delle GPU AMD

Questa sezione è basata principalmente sull'AMD OpenCL Univesity Kit e su [AMD, 2010c]. Si prende ad esempio la GPU Radeon HD 5870. Le GPU della serie "Cypress" sono realizzare secondo un'architettura a tre liveli. La Radeon HD 5870 infatti dispone di 1600 *stream processors* o *processing elements*, raggruppati a loro volta in 320 *stream cores*, a loro volta raggruppati in 16 SIMD engine (*compute unit* secondo la terminologia OpenCL); questa GPU contiene quindi 20 unità di elaborazione, ognuna delle quali contiene 16 *core* ognuno dei quali contiene 5 *processing element*. Le *compute unit* sono le unità di più alto livello a cui un processore di dispacciamento (denominato *Ultra-Threaded Dispatch Processor* nella documentazione AMD) contenuto nella GPU distribuisce i diversi kernel di esecuzione, all'interno di esse i diversi *stream core* si occupano di eseguire i kernel, ognuno dei quali esegue operazioni su un flusso di dati indipendente da quello degli altri e contiene i *processing element* che possono realizzare operazioni matematiche su interi, in virgola mobile a singola e doppia precisione e funzioni matematiche trascendenti. Ogni *stream core* è realizzato quindi come un processore VLIW (Very Long Instruction Word) a 5 vie che può eseguire fino a 5 operazioni scalari in un'istruzione VLIW (ognuna delle quali è eseguita su un corrispondente *processing element*) così composto:

- una *branch execution unit* per il controllo del flusso di esecuzione
- 5 *processing elements* che possono eseguire operazioni su interi e in virgola mobile a precisione singola, uno di questi denominato T-processing element può eseguire anche operazioni matematiche trascendenti (logaritmi e funzioni trigonometriche). Le operazioni in virgola mobile in doppia precisione devono essere eseguite utilizzando due dei quattro *processing elements* (escludendo quello che può effettuare operazioni trascendenti) o tutti e quattro
- registri *general purpose*

Gli *stream core* all'interno di una singola *compute unit* eseguono la stessa istruzione VLIW, il blocco di work-item che viene eseguito insieme su una *compute unit* è chiamato *wave-front* (è di 64 per la serie Cypress a cui appartiene

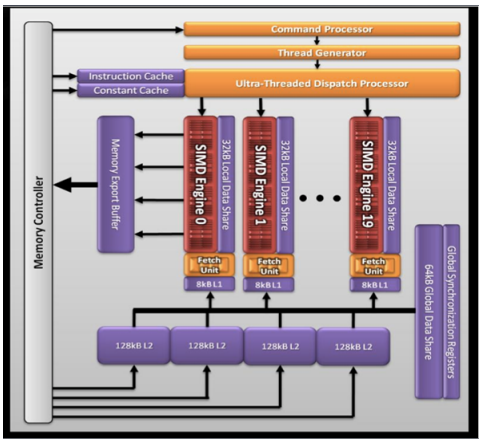


Figura 3.12: Architettura Cypress, struttura della GPU

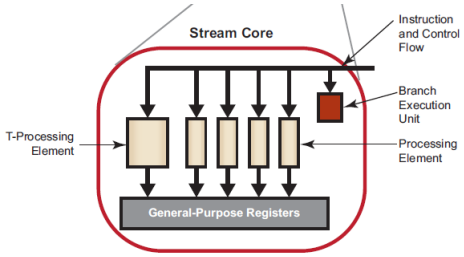


Figura 3.13: Struttura di un AMD *stream core*

la Radeon HD 5870). Ogni work-item è eseguito su un singolo stream core (servono infatti 4 cicli per eseguire un *wave-front,* 16 work-item ad ogni ciclo), ogni work-group è invece eseguito su una singola *compute unit* (SIMD engine) e non può esere distribuito su più unità, questo è infatti richiesto dalla specifica OpenCL, ogni *compute unit* può invece eseguire più work-group.

L'architettura della memoria in un SIMD Engine è così composta:

- Registri
- Local Data Store: memoria condivisa on-chip
- Cache L1 di 8KB

A queste si aggiunge una cache L2 di 512 KB condivisa tra diversi SIMD Engine. Sono disponibili due diversi tipi di percorsi per il trasferimento dei dati: Fast Path per le operazioni a 32 bit e Complete Path per le operazioni atomiche o su dati di larghezza inferiore a 32 bit.

La memoria dal punto di vista OpenCL è del tutto identica a quella esposta per la specifica OpenCL infatti OpenCL è uno standard multipiattaforma e non

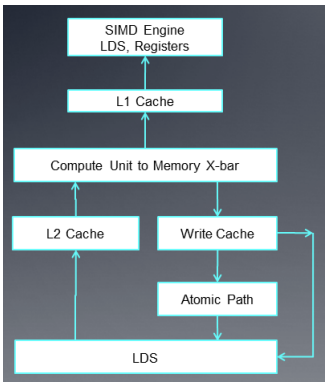


Figura 3.14: Architettura della memoria AMD per più SIMD Engine

deve dipendere dai singoli produttori, sarà poi il *vendor* ad associare i diversi livelli di memoria OpenCL alla reale architettura della propria piattaforma. Per AMD i dati condivisi tra più work-item (la *memoria locale* di OpenCL) utilizzano la memoria Local Data Share (per aumentare le performance sfruttando l'ampia banda passante verso ogni SIMD engine), la memoria privata utilizza i registri e la memoria costante utilizza la cache L1. La memoria costante beneficia nell'implementazione AMD di schemi di indirizzamento diretto per costanti non di tipo vettore (l'indirizzo è noto inizialmente) e di schemi di acesso allo stesso indice quando diversi work-item accedono alla stessa costante, gli array costanti globali possono inoltre utilizzare la cache L1 quando occupano meno di 16KB (ad esempio un vettore di 4096 dati in singola precisione). I casi in cui ogni item accede a indirizzi differenti di memoria costante non utilizzano la cache e offrono le stesse performance della memoria globale.

3.6.2 Architettura CUDA delle GPU Nvidia Fermi

Questa sezione è basata principalemente sull'AMD OpenCL Univesity Kit che utilizza "NVIDIA's Next Generation CUDA Architecture" come fonte. Si prende ad esempio la GPU GTX 480 che è costituita da 15 streaming multiprocessor (SM) o "core" costituiti ognuno da 32 CUDA core o "CUDA processors" (per un totale di 480). La memoria globale implementa un ECC (Error-Correcting Code, codice di correzione degli errori).

Gli SM eseguono ognuno gruppi di 32 thread denominati *warp*, poiché ogni SM contiene due scheduler e due unità di dispacciamento riesce ad eseguirne due alla volta, è chiaro quindi che l'esecuzione contemporanea aumenta l'efficienza (in particolare si riesce a sfruttare un nuovo core ad ogni 64 thread, work-item in OpenCL, in più). Il core CUDA è costituito da una singola ALU e un'unità in virgola mobile (FPU, Floating Point Unit).

L'esecuzione è di tipo SIMT (cioè SPMD su hardware SIMD), le istruzioni sono infatti scalari ma poiché tutti i thread condividono lo stesso flusso di istru-

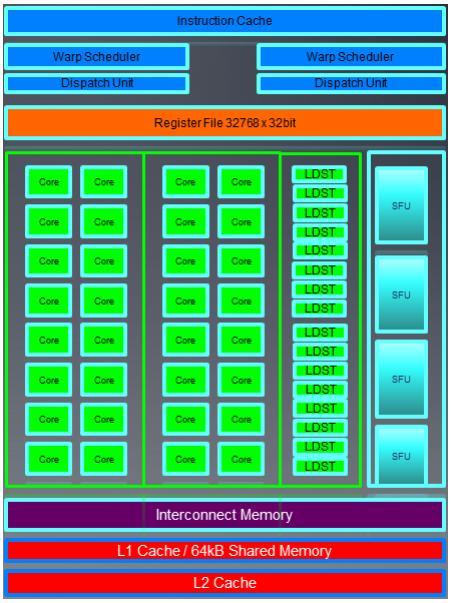


Figura 3.15: Architettura CUDA Fermi, struttura di uno Stream Multiprocessor

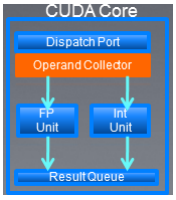


Figura 3.16: Struttura di un core CUDA

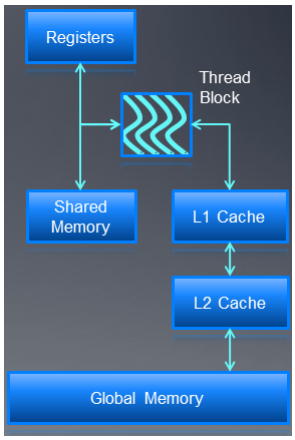


Figura 3.17: Gerarchia di memoria Nvidia per un singolo SM e comunicazione con memoria globale

zioni ogni SM può eseguire 32 operazioni identiche alla volta (grazie ai 32 CUDA core, ognuno con la propria ALU) su dati diversi in modo SIMD, l'esecuzione non è comunque di tipo SIMD poiché i diversi *warp* non sono sincronizzati, il kernel da eseguire specifica il comportamento (controllo di flusso) di un sngolo thread (senza che sia necessario preoccuparsi della sincronizzazione) e non è necessario esporre la larghezza del vettore (come per le istruzioni SSE). Il modello di esecuzione SIMT può essere combinato con il *pipelining:* dividendo un'istruzione in fasi e permettendo alla ALU di eseguire contemporanemente diverse fasi è possibile iniziare a eseguire una fase di un'istruzione e contemporaneamente la fase seguente di un'istruzione seguente (se non è legata al risultato della precedente).

La gerarchia di memoria Nvidia in un singolo SM è così composta:

- Registri per 32KB
- Cache L1 di 64KB per ogni SM configurabile per supportare memoria condivisa e caching della memoria globale (si può scegliere di associare 16KB alla memoria condivisa e 48KB alla cache della memoria globale o viceversa).
- Cache L2 (768KB) per servire tutte le operazioni (*load*, *store* e *texture*), *path* unificato a quello globale per *load* e *store*.

La memoria dal punto di vista OpenCL è del tutto identica a quella esposta per la specifica OpenCL anche se tratta dalla documentazione AMD, infatti OpenCL è uno standard multipiattaforma e non deve dipendere dai singoli produttori, sarà poi il *vendor* ad associare i diversi livelli di memoria OpenCL alla reale architettura della propria piattaforma. Per Nvidia i dati condivisi tra più work-item (la *memoria locale* di OpenCL) utilizzano la memoria condivisa (per evitare la latenza della memoria globale) e la memoria privata utilizza i registri.

Capitolo 4

Implementazioni con OpenCL

In questo capitolo dopo aver prima descritto le differenze generali delle implementazioni OpenCL rispetto al codice sequenziale si descriveranno nel dettaglio il codice (*host* e *kernel*) della prima implementazione OpenCL realizzata, le diverse modifiche introdotte per migliorare le prestazioni, ulteriori modi per aumentare le prestazioni ma non utilizzati (in particolare perché non applicabili a questa implementazione), l'implementazione Intel con OpenCL dello stesso problema e infine alcune considerazioni per l'aggiunta di un output grafico al programma.

4.1 Differenze generali

Le implementazioni fatte con OpenCL sone basate sulla traduzione in C del codice Fortran disponibile riportata in 2.6 con tre differenze rilevanti:

- le matrici sono ora puntatori a double (o float) il cui accesso viene fatto con un unico indice, sono tuttavia ancora interpretate come matrici, infatti l'indice viene ogni volta composto sulla base dei due indici e il vettore è pensato come una matrice le cui righe sono affiancate e considerando il primo indice come riga e il secondo come colonna questo si traduce nell'accedere a un elemento di un vettore `v` con `v[i*(numero_di_colonne)+j]`. Nel codice il numero di colonne è *Ny+2* che è stato associato alla variabile `width` poiché per il passaggio di parametri al kernel è sempre richiesto l'indirizzo della variabile passata e non si può ricavare l'indirizzo di un'operazione con il carattere `&` come si può fare con una variabile.

- nel codice *host* sono presenti le parti relative all'inizializzazione del codice OpenCL, alla preparazione dei kernel e all'avvio degli work-item che li eseguono

- la parte centrale del codice è stata spostata in una funzione di tipo *kernel* per poterla parallelizzare e nel codice host sono quindi presenti la creazione

dei kernel, l'assegnamento dei loro parametri e l'inserimento in una coda dei kernel (*command queue*) che corrisponde all'esecuzione degli work-item.

4.2 Codice host

Per realizzare un'applicazione OpenCL, come descritto nel capitolo 3.4, è necessario un *codice host* che è un normale programma C che utilizza tipi e funzioni contenuti nella libreria *cl.h* per costruire i kernel, avviare i rispettivi wok-item e costruire dati di tipi compatibili con i parametri richiesti dalle funzioni di tipo kernel. È comunque possibile utilizzare altri linguaggi con *binding* appositamente programmati, lo standard OpenCL prevede comunque la specifica solo per C e per un binding C++ ed NVIDIA fornisce nel suo SDK un binding C++ che non segue la specifica OpenCL.

Il programma offre all'utente tre scelte:

- si chiede se si vuole eseguire il programma su CPU invece che su GPU, questo è possibile perché la creazione del contesto di esecuzione e la compilazione dei kernel (il codice su cui si basa l'esecuzione parallela degli work-item) è fatta a tempo di esecuzione sulla base del dispositivo scelto tra quelli restituiti da una funzione apposita che richiede anche il tipo di dispositivo che si intende utilizzare (e i valori possibili sono *CL_DEVICE_TYPE_GPU* o *CL_DEVICE_TYPE_CPU* dall'ovvio significato)

```
f (answer == 'y' || answer =='Y')
{
  dev = CL_DEVICE_TYPE_CPU;
}
else
{
  dev = CL_DEVICE_TYPE_GPU;
}
```

- si chiede se si vuole calcolare ad ogni passo la massa (per verificare quanto il programma la conservi), in una versione iniziale questo rallentava il programma perché introduceva una forma di sincronizzazione (mutua esclusione) poiché altrimenti si sarebbero potuti perdere degli aggiornamenti (due work-item diversi avrebbero potuto leggere lo stesso valore della massa, aggiornarlo e quindi scrivere perdendo il contributo del primo work-item). Nelle versioni definitive si è aggirato il problema introducendo un vettore in cui ogni elemento rappresenta la massa di una cella che viene aggiornato durante il resto dei calcoli (solo se richiesto dall'utente) e poiché ogni work-item opera su una cella diversa non sorgono conflitti, la somma viene fatta poi in maniera sequenziale in un secondo tempo solo

se l'utente ha scelto di calcolare la massa. Il calcolo della massa non era invece problematico nella versione sequenziale.

- come per la versione sequenziale si chiede se si vuole cambiare la discretizzazione temporale calcolata automaticamente dal programma

La struttura generale della funzione *main* del programma è:

- dichiarazione delle variabili: oltre alle normali variabili si sono dovuti dichiarare dei *buffer* OpenCL (cioè variabili di tipo *cl_mem*) corrispettivi alle variabili contenenti i dati su cui operare, inoltre i dati non sono più rappresentati da vettori di tipo *double* (o *float*) ma è stato utilizzato il tipo *cl_double* (o *cl_float*) fornito dalla libreria *cl.h*
- lettura da un file di input di dati come dimensione del dominio, discretizzazione spaziale, tempo finale della computazione (cioè quantità di tempo da simulare), *plotstep* (è utilizzato per l'output su file, il nome indica sia il fatto che i file sono in un formato che il programma Paraview può utlizzare per generare grafici sia il fatto che la stessa variabile potrà essere utilizzata per decidere in quale momento generare un'immagine da visualizzare a schermo in caso si usi una visualizzazione in tempo reale) e *mannning*
- preparazione della piattaforma: OpenCL richiede un *runtime,* come spiegato nella specifica, in questo caso viene recuperato il primo disponibile (si richiede una *platform*, ogni *vendor* è tenuto a fornire piattaforme per la propria implementazione, attualmente AMD, Nvidia e Intel forniscono ognuno una sola piattaforma per la propria implementazione) tuttavia si potrebbe cercare quello di un particolare *vendor* se desiderato, dopo averlo ottenuto si richiede con *clGetDeviceIDs* la lista di dispositivi (potrebbero esserci più GPU o CPU supportate dal runtime come due GPU dello stesso produttore) di tipo CPU o GPU sulla base della scelta dell'utente memorizzata nella varabile *dev* (il runtime potrebbe supportare entrambe, è comunque necessario specificare il tipo di dispositivo voluto anche nel caso ce ne fosse solo uno supportato dall'implementazione *runtime* OpenCL in esecuzione). Attualmente il modello dei driver non permette di utilizzare GPU di diversi *vendor* contemporaneamente. Vengono recuperate dal dispositivo anche informazioni utili a dimensionare la computazione come il numero di *compute units* disponibili e le dimensioni globali e locali massime (il numero massimo di work-item eseguibili in totale o in un work-group)

```
/*Fine della preparazione e inizio  della
    preperazione OpenCL e calcoli*/
clGetPlatformIDs( 1, &platform, NULL );
clGetPlatformInfo(platform, CL_PLATFORM_NAME,
    sizeof(name), name, NULL);
printf("The␣name␣of␣the␣current␣platform␣is:␣%s\n"
    , name);
```

```
clGetPlatformInfo(platform, CL_PLATFORM_VENDOR,
    sizeof(name), name, NULL);
printf("The␣name␣of␣the␣vendor␣of␣the␣current␣
    platform␣is:␣%s\n", name);
cl_device_id device=NULL;
cl_context context;
cl_command_queue queue;
cl_program program;
cl_kernel calculate, copy;
cl_mem Qx_buf, Qx_old_buf, Qy_buf, Qy_old_buf,
    Z_buf, Z_old_buf, Zb_buf, mass_buf, sem_buf,
    temp_buf;
// Find the device
clGetDeviceIDs(platform, dev, 1, &device, NULL);
cl_uint compute_units;
size_t  global_work_size[2];
size_t  local_work_size;
size_t  num_groups;
/*Informazione sul numero di compute units potrà
    essere utile per scegliere numero di work-items
     e work-group*/
clGetDeviceInfo(device, CL_DEVICE_NAME, sizeof(
    name), &name, NULL);
printf("The␣name␣of␣the␣current␣device␣is:␣%s\n",
    name);
clGetDeviceInfo(device,
    CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(cl_uint), &
    compute_units, NULL);
printf("Number␣of␣compute␣units:␣%d\n",
    compute_units);
size_t num_items;
clGetDeviceInfo(device,
    CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(size_t),
    &num_items, NULL);
printf("Max␣number␣of␣work-items␣in␣a␣work-group␣
    is:␣%d\n", (cl_uint)num_items);
cl_uint dim;
clGetDeviceInfo(device,
    CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, sizeof(
    size_t), &dim, NULL);
printf("Max␣number␣of␣dimensions␣for␣work-items␣in
    ␣a␣work-group␣is:␣%d\n", dim);
size_t *sizes = (size_t*)malloc(dim * sizeof(
    size_t));
clGetDeviceInfo(device,
    CL_DEVICE_MAX_WORK_ITEM_SIZES, dim * sizeof(
```

```
    size_t), sizes, NULL);
printf("Max␣size␣for␣each␣dimension␣of␣work-items␣
    in␣a␣work-group␣are:");
for(int i=0; i<dim;i++)
{
  printf("␣%d", (cl_int)sizes[i]);
}
printf("\n");
global_work_size[0] = Nx;
global_work_size[1] = Ny;
```

- creazione di un *contesto* e di una *command queue*, cioè una coda per l'esecuzione degli work-item, associata al contesto creato e al dispositivo scelto (la coda crea un legame tra questi). Il contesto è associato ad una piattaforma e a un dispositivo, infatti il primo parametro NULL di *clCreateContext* indica la piattaforma, è necessario indicarla esplicitamente solo nel caso ci siano diverse implementazioni o l'implementazione la richieda esplicitamente per questa funzione, altrimenti viene scelta quella di default. La *command queue* è associata al contesto creato e a un device, è chiaro da questo che nel caso si vogliano utilizzare più dispositivi è necessaria una coda separata per ognuno di essi

```
// Create a context and command queue on that
    device.
//qualche implementazione potrebbe volere che
    venga indicata la runtimepiattaforma (
    implementazione di OpenCL)
context = clCreateContext(NULL, 1, &device, NULL,
    NULL, &ret);
queue = clCreateCommandQueue(context, device, 0, &
    ret);
printf("Context␣and␣command␣queue␣initialized\n");
// Minimal error check.
if( queue == NULL )
{
  printf("Compute␣device␣setup␣failed\n");
  return(-1);
}
```

- compilazione a tempo di esecuzione dei *kernel:* viene caricato in una stringa (con una funzione creata appositamente che legge in blocco i caratteri contenuti in un file) il testo sorgente delle funzioni kernel dopodiché viene creato un *programma* associato al sorgente e a un contesto (il quale è a sua volta associato a un particolare dispositivo del tipo scelto e a una piattaforma) che è successivamente compilato con *clBuildProgram*. Sulla

base del programma si possono poi creare i *kernel* cioè i codici compilati che potranno essere eseguiti dagli work-item. Il kernel dopo essere stato creato può essere utilizzato più volte con parametri diversi

```
// Perform runtime source compilation
char *source = load_program_source("kernel.cl");
printf("Source␣loaded\n");
program = clCreateProgramWithSource(context, 1, (
    const char **) &source, NULL, NULL);
ret = clBuildProgram( program, 1, &device, NULL,
    NULL, NULL );
//Print compiler error messages
if(ret != CL_SUCCESS)
{
  printf("clBuildProgram␣failed:␣%d\n", ret);
  char buf[0x10000];
  clGetProgramBuildInfo( program, device,
      CL_PROGRAM_BUILD_LOG, 0x10000, buf, NULL);
  printf("\n%s\n", buf);
  return(-1);
}
printf("Creating␣kernels...\n");
//Create kernels entry point
calculate = clCreateKernel( program, "calculate",
    NULL);
printf("Kernels␣created\n");
```

- inizializzazione dei buffer OpenCL: tutti i buffer dovranno essere riscritti, alcuni sono creati vuoti, altri (ad esempio quelli contenenti l'altezza dell'acqua nei diversi punti al tempo zero nella computazione) devono essere inizializzati. Si riporta a titolo di esempio una variabile per ogni caso: il buffer corrispondente a Qx, in cui si nota l'opzione *CL_MEM_COPY_HOST_PTR* (che indica che il contenuto è copiato da un puntatore della memoria host, in OpenCL i trasferimenti di memoria tra *host* e *device*, sono espliciti)

```
//Qx_old, Qy_old, Z_old, sem sono già inzializzati
     e saranno utilizzati più volte in lettura e
    scrittura
Qx_old_buf = clCreateBuffer(context,
    CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, (Nx
    +2)*(Ny+2)*sizeof(cl_double), Qx_old, NULL);
```

e il buffer corrispondente a Qx_old che è creato vuoto

```
//Qx, Qy, Z e mass sono vuoti e saranno utilizzati
    più volte in lettura e scrittura
Qx_buf = clCreateBuffer(context, CL_MEM_READ_WRITE
   , (Nx+2)*(Ny+2)*sizeof(cl_double), NULL, NULL);
```

- ciclo per i calcoli eseguito un numero di passi pari a `nstep = ceil(Time /dt)` dove *Time* è stato scelto dall'utente e *dt* calcolato come `dt = dx/ sqrt(9.8*(Zmax-Zmin))*0.1` oppure inserito dall'utente se ha scelto di cambiarlo in modo simile alla versione sequenziale, ogni passo del ciclo esegue le seguenti azioni:

  - copia di righe e colonne (nella prima versione del codice che utilizza OpenCL è eseguita comunque in modo seriale dalla CPU, è quindi identico al codice riportato in 2.6 per la versione C)
  - aggiornamento dei buffer sulla base delle modifiche dovute al passo precedente (in una versione successiva riportata in 4.5.1, poichè le copie di righe e colonne sono fatte direttamente da OpenCL, questo passo non è più necessario). A titolo di esempio si riporta il caso per *Z_old*

    ```
    clEnqueueWriteBuffer(queue, Z_old_buf, CL_TRUE,
        0, (Nx+2)*(Ny+2)*sizeof(cl_double), Z_old,
       0, NULL, NULL);
    ```

  - calcolo effettuato dagli work-item per ottenere i valori per il passo successivo, gli input sono buffer con un nome di tipo X_old e i risultati vengono scritti su buffer di tipo X dove la X è il nome della variabile come Z o Qx che rimane ugule nelle versione X e nel suo corrispettivo X_old (questa non è una caratteristica necessaria per OpenCL ma solo una convenzione utilizzata in questo codice per la nomenclatura). Nonostante la notazione estesa si tratta di una chiamata a funzione che aggiunge i parametri ad uno stack, come già indicato il kernel può essere utilizzato più volte con argomenti diversi, questo è del tutto analogo al riuso di una funzione. La chiamata `clEnqueueNDRangeKernel( queue, calculate, 2, NULL, global_work_size, NULL, 0, NULL, NULL)` è quella che avvia effetivamente gli work-item che eseguiranno nella coda indicata nel pimo parametro il codice del kernel indicato nel secondo e saranno in numero pari al quinto, il sesto parametro indica invece la dimensione locale cioè quella di un singolo work-group (qui è NULL, si sono realizzare versioni che invece utilizzano questo parametro, in particolare è *necessario* nel caso si voglia sfruttare la memoria locale ma è possibile comunque dividere gli item in diversi work-group). La chiamata finale a funzione `clFinish(queue)` introduce una forma di sincronizzazione, garantisce infatti che il codice host non prosegua

fino a che tutti gli work-item contenuti nella *command queue* non avranno terminato, la sola chiamata clEnqueueNDRangeKernel è infatti non bloccante

```
clSetKernelArg(calculate, 0, sizeof(Z_buf), (
    void*) &Z_buf);
clSetKernelArg(calculate, 1, sizeof(Z_old_buf),
     (void*) &Z_old_buf);
clSetKernelArg(calculate, 2, sizeof(Qx_buf), (
    void*) &Qx_buf);
clSetKernelArg(calculate, 3, sizeof(Qx_old_buf)
    , (void*) &Qx_old_buf);
clSetKernelArg(calculate, 4, sizeof(Qy_buf), (
    void*) &Qy_buf);
clSetKernelArg(calculate, 5, sizeof(Qy_old_buf)
    , (void*) &Qy_old_buf);
clSetKernelArg(calculate, 6, sizeof(Zb_buf), (
    void*) &Zb_buf);
clSetKernelArg(calculate, 7, sizeof(mass_buf),
    (void*) &mass_buf);
clSetKernelArg(calculate, 8, sizeof(dt), (void
    *) &dt);
clSetKernelArg(calculate, 9, sizeof(dx), (void
    *) &dx);
clSetKernelArg(calculate, 10, sizeof(width), (
    void*) &width);
clSetKernelArg(calculate, 11, sizeof(n), (void
    *) &n);
clSetKernelArg(calculate, 12, sizeof(sem_buf),
    (void*) &sem_buf);
clSetKernelArg(calculate, 13, sizeof(usemass),
    (void*) &usemass);
//avvio degli work-item
clEnqueueNDRangeKernel( queue, calculate, 2,
    NULL, global_work_size, NULL, 0, NULL, NULL)
    ;
/*aspetta la terminazione di tutti gli work-
    item prima di recuperare i risultati e
    scrivere,
* è necessario anche quando non si scrive
    perché il ciclo successivo potrebbe
    riniziare con valori vecchi*/
clFinish(queue);
```

– se richiesto, recupero della massa sommando tutti i valori memorizzati nel vettore apposito

– se e solo se ci si trova in un passo multiplo di *plotstep* recupero dei dati calcolati per l'output su file o la visualizzazione. Qui di seguito il codice per i recupero dei dati che sincronizza le normali variabili con il contenuto dei rispettivi buffer, si fa notare che il runtime OpenCL non ha cognizione di quali buffer corrispondano alle variabili ed è necessario indicare per ogni variabile "normale" di tipo vettore da quale buffer è necessario recuperare i dati

```
Z = (cl_double *) clEnqueueMapBuffer( queue,
    Z_buf, CL_TRUE, CL_MAP_READ, 0, (Nx+2)*(Ny
    +2)*sizeof(cl_double), 0, NULL, NULL, NULL )
    ;
Qx = (cl_double *) clEnqueueMapBuffer( queue,
    Qx_buf, CL_TRUE, CL_MAP_READ, 0, (Nx+2)*(Ny
    +2)*sizeof(cl_double), 0, NULL, NULL, NULL )
    ;
Qy = (cl_double *) clEnqueueMapBuffer( queue,
    Qy_buf, CL_TRUE, CL_MAP_READ, 0, (Nx+2)*(Ny
    +2)*sizeof(cl_double), 0, NULL, NULL, NULL )
    ;
```

– copia dei nuovi valori calcolati nei buffer X_old (i nuovi valori saranno i vecchi valori al passo successivo), i buffer normali rimangono uguali, saranno semplicemente riscritti nel passo successivo, non importa il loro contenuto

```
/* Copy data from new to old */
clEnqueueCopyBuffer(queue, Z_buf, Z_old_buf, 0,
     0, (Nx+2)*(Ny+2)*sizeof(cl_double), 0, 0,
    0);
clEnqueueCopyBuffer(queue, Qx_buf, Qx_old_buf,
    0, 0, (Nx+2)*(Ny+2)*sizeof(cl_double), 0, 0,
     0);
clEnqueueCopyBuffer(queue, Qy_buf, Qy_old_buf,
    0, 0, (Nx+2)*(Ny+2)*sizeof(cl_double), 0, 0,
     0);
```

– recupero dei dati, è necessario sincronizzare i dati poiché all'inizio del passo successivo del ciclo la copia viene fatta utilizzando le normali variabili e non i buffer OpenCL (questo passo è saltato nella versione in 4.5.1 accennata in precedenza in cui le copie di righe e colonne sono fatte dagli work-item). Il parametro *CL_TRUE* indica che l'operazione è bloccante e quindi la funzione non finisce fino a che l'operazione non è stata portata a termine, nel caso fosse stato *CL_FALSE* l'operazione sarebbe stata non bloccante e per usare

con sicurezza i puntatori resistuiti si sarebbe dovuto interrogare una variabile puntatore evento passata alla funzione come penultimo parametro.
In alternativa è possibile utilizzare la funzione *clEnqueueReadBuffer* che ha lo stesso effetto di *clEnqueueMapBuffer* con l'opzione CL_READ ma non sono state notate differenze di prestazioni inoltre nelle versioni successive utilizzanti la memoria locale la chiamata di queste funzioni è necessaria solo per l'output e non ha inciso sui risultati dei test riportati (che sono stati effettuati senza output)

```
//è necessario recuperare dati perché copia di
    righe/colonne a inizio ciclo è fatta sui
    valori normali e non dal kernel coi buffer
Z_old = (cl_double *) clEnqueueMapBuffer(queue,
     Z_old_buf, CL_TRUE, CL_MAP_READ, 0, (Nx+2)
    *(Ny+2)*sizeof(cl_double), 0, NULL, NULL,
    NULL );
Qx_old = (cl_double *) clEnqueueMapBuffer(
    queue, Qx_old_buf, CL_TRUE, CL_MAP_READ, 0,(
    Nx+2)*(Ny+2)*sizeof(cl_double), 0, NULL,
    NULL, NULL ); Qy_old = (cl_double *)
    clEnqueueMapBuffer( queue, Qy_old_buf,
    CL_TRUE, CL_MAP_READ, 0, (Nx+2)*(Ny+2)*
    sizeof(cl_double), 0, NULL, NULL, NULL );
```

4.3 Parallelizzazione

Poiché ad ogni passo ogni valore della superficie (la matrice Z) e delle quantità (la matrice Qx) è calcolato solo sulla base di valori di Z e Qx al passo precedente è possibile eseguire tutti questi calcoli in modo parallelo (non c'è nessun calcolo allo stesso passo che dipenda da valori calcolati durante lo stesso passo per un'altra cella, cioè un'altra coppia di coordinare, è quindi possibile avviare uno work-item per ogni coppia di coordinate). Qui di seguito si riporta il codice della funzione kernel *calculate* nella versione più semplice. In OpenCL le funzioni di tipo kernel devono essere dichiarate con tipo di ritorno *void* e precedute dal modificatore *__kernel,* la struttura della funzione che utilizza parametri di tipo puntatore (anche questo richiesto dalla specifica) sia per l'input sia per l'output senza nessun valore restituito dalla funzione è dovuta quindi alla necessità di utilizzare questa struttura.

Il codice è estremamente simile a quello della versione sequenziale riportato in 2.6, le differenze sono le seguenti:

- il codice è spostato in una funzione inserita in un file diverso da quello del codice sorgente del programma host, questo è comodo in quanto consente di reperire il codice dei kernel che andrà compilato a tempo di esecuzione

Algoritmo 4.1 Codice C del kernel OpenCL per l'esecuzione parallela del metodo Lax-Friedrichs per le acque basse

```
1  __kernel void calculate(__global double* Z, __global double*
       Z_old, __global double* Qx, __global double* Qx_old,
      __global double* Qy, __global double* Qy_old, __global
      double* Zb, __global double* mass, const double dt, const
       double dx, const int width, const double n, __global int
      * sem, const int usemass)
2  {
3    double u, v, S; double Fpx, Fmx, Gpx, Gmx; double Fpy, Fmy
       , Gpy, Gmy;
4    //c'è +1 perché gli indici degli work-item vanno da 0 a Nx
       -1 o Ny-1 ma a noi interessano le posizioni da 1 a Nx o
       Ny
5    int i = get_global_id(0)+1;
6    int j =get_global_id(1)+1;
7    /*ogniwork-item è associato a una coppia (i, j)*/
8    /*Core code*/
9    Z[i*width+j]=0.25*(Z_old[(i+1)*width+j]+Z_old[(i-1)*width+
       j]+Z_old[i*width+j+1]+Z_old[i*width+j-1])-dt/(2.0*dx)*(
       Qx_old[(i+1)*width+j]-Qx_old[(i-1)*width+j]+Qy_old[i*
       width+j+1]-Qy_old[i*width+j-1]);
10   if(usemass)
11   {
12   mass[i*width+j]=Z[i*width+j]*dx*dx;
13   }
14   Fpx=Qx_old[(i+1)*width+j]*Qx_old[(i+1)*width+j]/Z_old[(i
       +1)*width+j]+0.5*9.8*Z_old[(i+1)*width+j];  Fmx=Qx_old
       [(i-1)*width+j]*Qx_old[(i-1)*width+j]/Z_old[(i-1)*width
       +j]+0.5*9.8*Z_old[(i-1)*width+j];  Gpx=Qx_old[(i+1)*
       width+j]*Qy_old[(i+1)*width+j]/Z_old[(i+1)*width+j];
15   Gmx=Qx_old[(i-1)*width+j]*Qy_old[(i-1)*width+j]/Z_old[(i
       -1)*width+j];
16   Qx[i*width+j]=0.25*(Qx_old[(i+1)*width+j]+Qx_old[(i-1)*
       width+j]+Qx_old[i*width+j+1]+Qx_old[i*width+j-1])-dt
       /(2*dx)*(Fpx-Fmx+Gpx-Gmx);
17   Fpy=Qx_old[i*width+j+1]*Qy_old[i*width+j+1]/Z_old[i*width+
       j+1];
18   Fmy=Qx_old[i*width+j-1]*Qy_old[i*width+j-1]/Z_old[i*width+
       j-1];
19   Gpy=Qy_old[i*width+j+1]*Qy_old[i*width+j+1]/Z_old[i*width+
       j+1]+0.5*9.8*(Z_old[i*width+j+1]);
20   Gmy=Qy_old[i*width+j-1]*Qy_old[i*width+j-1]/Z_old[i*width+
       j-1]+0.5*9.8*(Z_old[i*width+j-1]);
21   /*computation of friction term along y*/
22   u=Qx[i*width+j]/Z[i*width+j];
23   v=Qy[i*width+j]/Z[i*width+j];
24   S=(n*n)*v*sqrt(u*u+v*v)/pow(Z[i*width+j],4.0/3.0);
25   Qy[i*width+j]=0.25*(Qy_old[(i+1)*width+j]+Qy_old[(i-1)*
       width+j]+Qy_old[i*width+j+1]+Qy_old[i*width+j-1])-dt
       /(2*dx)*(Fpy-Fmy+Gpy-Gmy);
26 }
```

(l'alternativa sarebbe stata inizializzare nello stesso file del codice host una variabile con l'intero testo del codice e si sarebbe persa chiarezza)

- poiché il codice è trasferito in una funzione a parte tutte le variabili che prima potevano essere utilizzate direttamente devono essere ora passate come parametro alla funzione. In particolare ci sono in più la variabile width (che indica il numero di colonne), si sarebbe potuto anche utilizzare Ny e poi creare width (anche se nel caso sarebbe stato fatto da tutti gli work-item poiché questi sono eseguiti in modo parallelo) tuttavia questo valore è necessario anche nel codice host e l'uso di una variable (al posto di Ny+2) aumenta la chiarezza, diminuisce l'uso delle parentesi e risparmia addizioni

  - come già riportato si può notare che molti parametri sono di tipo puntatore, nonostante i parametri siano di tipo array l'utilizzo di puntatori non è dovuto a questo ma solo al fatto che ciò è richiesto dalla specifica OpenCL per i parametri di tipo *__global,* sarebbe quindi stato necessario anche per valori singoli. Come descritto in 3.4 il meccanismo della memoria globale è quello che permette di comunicare con i dati dell'*host*. I parametri di tipo *const* non devono essere invece di tipo puntatore e, come ovvio dal nome, non possono essere modificati all'interno del kernel

- non ci sono più i cicli for sugli indici infatti nel codice host sono creati tanti work-item quante sono le celle e ognuno di essi è associato ad una coppia di coordinate (come già spiegato con OpenCL è necessario usare vettori per rappresentare matrici tuttavia gli work-item possono essere identificati da coordinate multidimensionali) e quindi la cella è determinata automaticamente dagli identificatori globali degli work-item. Come è usuale in C si conta a partire da 0 e scegliendo quindi come dimensioni Nx ed Ny gli indici che indentificano gli work-item sono compresi tra (0, 0) e (Nx-1, Ny-1), tuttavia per come sono strutturate le matrici nel codice host è necessario inserire valori nelle celle da (1,1) a (Nx, Ny), quindi per allineare gli indici viene aggiunto 1 ogni volta che viene richiesto l'identificatore. Gli work-item sono così associati alle celle da (1, 1) a (Nx, Ny) e alle celle restanti (con indice 0 o Nx+1 in una dimensione e 0 o Ny+1 nell'altra) che costituiscono una cornice intorno a quelle non è associato nessun item, i valori contenuti sono però utilizzati dagli item corrispondenti alle celle adiacenti (infatti ogni item utilizza valori delle 4 celle adiacenti a quella in cui scriverà il risultato). Si fa notare che anche le dimensioni sono contate a partire da 0 e quindi la prima dimensione è identificata da 0 e la seconda da 1 come si può notare dal codice host in 4.2 che stampa il valore massimo per ogni dimensione.

- La variazione di massa non viene più immediatamente sommata alla variabile di controllo ma inserita in una cella di un vettore che, come spiegato

nella sezione 4.2, verrà sommato nel caso si sia richiesto il controllo della massa

4.4 Caratteristiche dipendenti dalla piattaforma

Si è scelto di utilizzare variabili in virgola mobile a doppia precisione come si usa solitamente nelle applicazioni scientifiche tuttavia il supporto ai calcoli a doppia precisione è opzionale nelle specifiche di OpenCL 1.0 e 1.1 e infatti non è supportato dall'implementazione OpenCL di AMD/ATI ma lo è dalla soluzione di Nvidia, tuttavia l'ATI Stream SDK v2.2 fornisce un'estensione proprietaria in alternativa.

Per utilizzare i calcoli in doppia precisione è necessario inserire nel file che contiene i kernel la seguente direttiva di preprocessore (standard) per l'esecuzione su implementazioni Nvidia di OpenCL:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
```

e la seguente (proprietaria) per le implementazioni AMD/ATI

```
#pragma OPENCL EXTENSION cl_amd_fp64 : enable
```

Queste direttive non sono necessarie nel caso si usi il tipo *cl_float* che non è una parte opzionale dello standard e quindi *deve* essere disponibile in ogni implementzione di OpenCL

4.5 Modifiche introdotte

Dopo i primi di test (i risultati sono presentati nel capitolo successivo) ci si è accorti che nelle configurazioni hardware disponibili nonostante un grande miglioramento di prestazioni rispetto alla versione sequenziale non emergeva una chiara differenza di prestazioni a favore della GPU, come atteso, e si è quindi tentato di migliorare le prestazioni apportando alcune modifiche al codice.

4.5.1 Copia parallela di righe e colonne

Nella versione originale del codice vi è un punto in cui è necessario copiare righe o colonne della matrice in quelle adiacenti, questo viene fatto in modo del tutto sequenziale. Si è realizzata una versione in cui questa parte è spostata in un kernel OpenCL affinché le copie avvenissero in maniera parallela (sia su CPU sia su GPU). Per fare ciò si sono modificate le funzioni che effettuano tali copie in modo che avviassero la copia in parallelo, il codice è molto simile a quello descritto nella sezione 4.2 con l'ovvia differenza che il kernel utilizzato è diverso e che il numero totale di work-item da eseguire (la dimensione globale) è pari al numero di celle da copiare invece che al numero totale di celle della matrice utilizzata. Qui di seguito si riporta il codice del kernel utilizzato per la copia delle righe, come si può notare la dimensione globale utilizzata è una sola e coincide con l'indice j che varia sulle diverse colonne (il secondo indice) della

singola riga (l'indice *row1* passato come parametro non modificabile al kernel). Non sono stati effettuati ulteriori miglioramenti poiché si tratta di un semplice trasferimento di dati (in particolare una copia) che deve obbligatoriamente utilizzare la memoria globale in ingresso e in uscita quindi nel caso di esecuzione su GPU non si possono ridurre i trasferimenti e l'utilizzo di memoria privata o locale o altri metodi avrebbero appesantito inutilmente il codice senza ridurre gli accessi alla memoria globale.

Qui si riporta l'esempio della funzione *copyrow* che ha parametri simili alla corrispettiva funzione sequenziale, sono tuttavia necessari anche la coda di esecuzione e il kernel da utilizzare per poter avviare gli work-item, un'altra scelta sarebbe potuta essere la creazione di *command queue* e *kernel* specifici all'interno di ogni funzione ma non c'era motivo di farlo poiché queste funzioni devono essere avviate tra un ciclo di calcolo e l'altro possono quindi condividere la stessa coda di esecuzione e i *kernel* possono essere riutilizzati identici dopo la prima creazione da invocazioni diverse della stessa funzione e anche da diverse funzioni.

```
void copyrow(cl_command_queue queue, cl_kernel kernel,
    cl_mem mat1, int row1, cl_mem mat2, int row2, int
   size1, int size2) {
        size_t  global_work_size[1];
        global_work_size[0] = size2;
        clSetKernelArg(kernel, 0, sizeof(mat1), (void
           *) &mat1);
        clSetKernelArg(kernel, 1, sizeof(row1), (void
           *) &row1);
        clSetKernelArg(kernel, 2, sizeof(mat2), (void
           *) &mat2);
        clSetKernelArg(kernel, 3, sizeof(row2), (void
           *) &row2);
        clSetKernelArg(kernel, 4, sizeof(size2),(void
           *) &size2);
        clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
            global_work_size, NULL, 0, NULL, NULL);
        clFinish(queue);
}
```

Qui di seguito invece il riferimento alla funzione *kernel* copyrow i cui parametri sono del tutto simili alla funzione *copyrow* sequenziale

```
__kernel void copyrow(__global double* mat1, const int
    row1, __global double* mat2, const int row2, const
    int size2) {
                int j = get_global_id(0);
                mat1[row1*size2+j] = mat2[row2*size2+j
                   ];
}
```

che è passato come secondo parametro alla funzione *copyrow* precedente dopo essere stato costruito nel seguente modo

```
copyrowkern = clCreateKernel( program, "copyrow", NULL
    );
```

allo stesso modo in cui è costruito il kernel *calculate*.

Una prima versione utilizzava un numero di work-item pari all'intero numero delle celle della matrice utilizzata controllando poi quali fossero realmente corrispondenti alle celle da copiare tuttavia, nonostante una differenza di prestazioni quasi nulla, si è deciso di semplificare il codice nella versione riportata per mantenerlo più semplice e pulito.

Si fa notare che utilizzando una copia parallela tramite OpenCL e non tramite normale codice host si elimina la necessità di sincronizzazione in due punti:

- non è necessario scrivere i nuovi valori dovuti alle copie di righe e colonne nei buffer di tipo *cl_mem*, i buffer che saranno usati come parametro per i kernel OpenCL successivi contengono già i valori corretti a causa dell'esecuzione del kernel precedente
- a fine ciclo non è necessario recuperare i dati cacolati dagli work-item che hanno eseguito il kernel *calculate* poiché le copie di righe e colonne all'inizio del ciclo successivo saranno effettuate sui buffer OpenCL e non sulle variabili host

4.5.2 Uso della memoria locale

Nella prima versione OpenCL uno dei problemi è il continuo accesso alla memoria globale: come si può notare dal codice del kernel in 4.3 ogni volta che è necessario un valore viene utilizzato uno dei parametri etichettato *__global* della funzione kernel *calculate*. Questo significa che ogni volta che è necessario un parametro del kernel esso deve essere recuperato dalla memoria globale. Come spiegato in 3.4 lo standard OpenCL permette l'utilizzo di una memoria locale nel dispositivo utilizzato che può essere acceduta da più work-item (nel caso della GPU è una memoria comune a più unità di elaborazione della GPU, denominata *local data share* nel caso AMD). Utilizzare una memoria locale inizializzata da un certo work-item con i valori necessari allo stesso riduce il numero di accessi alla memoria globale alla sola copia iniziale e non più al numero di occorrenze della variabile.

Al contrario della prima implementazione OpenCL riportata in 4.2 il numero di work-item non coincide con il numero totale di celle escluso il bordo (ci sono quindi item anche per il bordo più esterno) e ogni work-item all'inizio della sua computazione copia un valore della memoria globale nella cella della memoria locale a lui idealmente corrispondente (questo è realizzato con l'operatore resto, supportato da OpenCL come le altre operazioni aritmetiche) infatti si è scelto di utilizzare a livello globale un numero di work-item pari almeno al numero di celle e a livello locale dimensionare il numero di work-item sul numero di celle

della memoria locale (si fa notare che la dimensione locale di esecuzione, cioè il numero di work-item eseguiti localmente, e la dimensione della memoria locale, cioè il numero di celle delle variabili locali moltiplicato per la dimensione di ogni cella, sono due cose diverse così come lo sono il numero globale di work-item e la dimensione dei dati su cui essi devono operare). Il numero di work-item è pari al numero totale di celle tuttavia la memoria globale deve essere un multiplo di quella locale, il numero di celle globali è per questo forzato su ogni dimensione al primo multiplo della corrispondente dimensione locale superiore alla dimensione globale che sarebbe richiesta, ad esempio una griglia 1000x1000 che richiederrebe 1002x1002 celle diventa 1008x1008 se la dimensione locale è 16 per entrambe le dimensioni, e ogni cella avrà un work-item associato, in seguito il kernel controllerà se il work-item in esecuzione è utile e avvierà i calcoli solo in quelli che devono contribuire realmente al risultato. Per mantenere semplice il codice non si è gestito esplicitamente il caso di una dimensione già multipla e la scelta ricade comunque sul primo multiplo superiore quindi se ad esempio la dimensione locale di una griglia con lato 1000 è 1 (in versioni successive è possibile scegliere dimensioni arbitrarie) si ottiene una dimensione globale di 1003. Qui di seguito il codice utilizzato per decidere la memoria globale dopo la scelta della dimensione locale (utilizzando la radice delle dimensioni massime o un numero arbitrario), nel caso la dimensione locale sia superiore alla dimensione richiesta (nel codice riportato è Nx+2) si utilizza come dimensione globale una dimensione equivalente a quella locale, cioè si utilizza un solo work-group.

```
if(Nx+2 < local_work_size[0])
{
        global_work_size[0] = local_work_size[0];
}
else
{
        global_work_size[0] = ((Nx+2)/local_work_size[0]+1)*
            local_work_size[0];
}
```

Si sono realizzate due versioni. Nella prima, dopo aver fatto la copia in memoria locale, il codice controlla se lo work-item in esecuzione è associato a una casella vicino al bordo della memoria locale oppure no, nel primo caso si recupera dalla memoria globale i dati non disponibili che dovranno essere utilizzati (cioè i dati dalle caselle adiacenti a quelle del bordo che non sono disponibili dalla memoria globale) e inseriti in una variabile privata utilizzata poi per i calcoli veri e propri altrimenti la stessa variabile viene inizializzata con dati dalla memoria locale. Nella seconda versione (qui descritta più dettagliatamente) gli work-item associati a caselle vicino al bordo di una "tessera" di memoria locale copiano anche i valori adiacenti (in questo caso la memoria locale è più grande per includere il nuovo bordo, si ricorda che la dimensione delle variabili locali può essere diversa dalla dimensione locale cioè dal numero di work-item in un work-group) in questo modo i diversi work-item possono operare normalmente come nel caso dell'accesso alla memoria globale senza debordare mai dalla memoria locale assegnata al loro work-group di appartenenza. In entrambe le

versioni si è scelto di utilizzare come dimensione locale la massima dimensione disponibile per il dispositivo che può essere ottenuta interrogandolo con funzioni OpenCL standard, l'unica accortezza è stata quella di fare in modo che il prodotto delle due dimensioni scelte (si ricorda che il codice lavora idealmente su matrici e gli work-item sono identificati da *due* indici) non fosse superiore alla dimensione totale consentita infatti le dimensioni massime possono coincidere, ad esempio per la serie Radeon HD 5000 il massimo è 256 sia per ogni dimensione sia per il totale di work-item in un work-group per cui è possibile ad esempio una struttura 16x1 (cioè una singola dimensione) o 128x2 ma non 265x265, inoltre il totale di dimensioni che è possiible usare è 3. Per la CPU Core i7 utilizzata invece il massimo possibile è sempre 1024 e il numero massimo di dimensioni utilizzabili è 3. Per la GTS450 invece i limiti sono di 1024 work-item in un work-group e il limite è lo stesso per le prime due dimensioni e solo di 64 per la terza (sarebbe quindi possibile ad esempio utilizzare un work-group 16x16x4, cosa non possibile su Radeon 5830 ma possibile su Core i7 930 nel caso non ci siano altre limitazioni). La Tesla C1060 può invece gestire una dimensione massima di 512 item in un gruppo e dimensioni massime di 512, 512 e 64 elementi per le rispettive dimensioni. Nei casi in esame si è scelto di effettuare la radice di ogni dimensione e poi verificare che il prodotto non superasse il totale massimo consentito così, a meno di indicazioni contrarie, si sono utilizzati work-group 16x16 per la GPU ATI utilizzata e 32x32 per la CPU Intel utilizzata per un totale rispettivamente di 256 e 1024. Per la GTS il codice ottiene una dimensione di 32x32 e per la Tesla una dimensione di 22x22 pari a una dimensione di 484. Nel caso la dimensione locale sia superiore a quella della dimensione del problema si utilizza come dimensione globale quella locale e non vengono effettuati calcoli relativi alle celle in eccesso, come si può notare nel codice riportato è presente il controllo **if**(i != 0 && j != 0 && i < Nx+1 && j < width−1) che serve anche nel caso la dimensione richiesta dal problema non sia esattamente un multiplo della dimensione locale (possibilità già accennata). Nella memoria host si controlla anche che la memoria locale (quella associata ad ogni work-group) non ecceda quella disponibile. La memoria locale è composta da registri a uso generico e da una memoria LDS (Local Data Store). Le GPU della serie 5800 hanno 256 KB di registri, questi sono realizzati in modo che siano accessibili da diversi wave-front (l'insieme minimo di work-item eseguibili, pari a 64) e ogni kernel alloca solo i registri di cui ha bisogno, al contrario di una CPU che alloca un numero fisso di registri. Le stesse GPU hanno 32 KB di LDS accessibile da diversi work-group eseguiti dalla medesima compute unit. Il numero di registri utilizzati e il LDS limitano il numero di wave-front eseguibili sulla singola compute unit. Ad esempio con 30 registri utilizzati dal kernel si possono eseguire al massimo 64x8 work-item su una compute unit e con meno di 4 KB di LDS per ogni work-group si possono eseguire al massimo 64x32 work-item su una compute unit cioè 8 work-group di 4 wave-front ciascuno (aumentando il numero di registri o la dimensione del LDS i valori massimi diminuiscono). È possbile anche fissare a tempo di compilazione la dimensione di default degli work-group.

Si riporta qui di seguito il codice della seconda versione implementata che

utilizza memoria locale.

Listing 4.1: Codice C del kernel OpenCL per l'esecuzione parallela del metodo Lax-Friedrichs per le acque basse con utilizzo di memoria locale e numero ridotto di accessi alla memoria dell'host

```
1  __kernel void calculate(
2          __global double* Z,
3          __global double* Z_old,
4          __global double* Qx,
5          __global double* Qx_old,
6          __global double* Qy,
7          __global double* Qy_old,
8          __global double* Zb,
9          __global double* mass,
10         const double dt,
11         const double dx,
12         const int width,
13         const double n,
14         __global int* sem,
15         const int usemass,
16         __local double* Z_old_loc,
17         __local double* Qx_old_loc,
18         __local double* Qy_old_loc,
19         const int Nx
20                                                  )
21 {
22         double u, v, S;
23         double Fpx, Fmx, Gpx, Gmx;
24         double Fpy, Fmy, Gpy, Gmy;
25         //indici degli work-item vanno da 0 a Nx+1 o Ny+1
26         int i = get_global_id(0);
27         int j = get_global_id(1);
28         /*ogni work-item è associato a una coppia (i, j)*/
29         int ls[2];
30         ls[0] = get_local_size(0);
31         ls[1] = get_local_size(1);
32         //indici per memoria locale
33         int il = i % ls[0];
34         int jl = j % ls[1];
35         ls[0] =ls[0]+2;
36         ls[1] =ls[1]+2;
37         //la dimensione locale non coincide per forza con la
               dimensione della MEMORIA locale, in questo caso
           va sommato 2
38         //trasferimento dati a memorie locali
39         Z_old_loc[(il+1)*ls[1]+jl+1] = Z_old[i*width+j];
40         Qx_old_loc[(il+1)*ls[1]+jl+1] = Qx_old[i*width+j];
41         Qy_old_loc[(il+1)*ls[1]+jl+1] = Qy_old[i*width+j];
42         if(il == 0)
43         {
```

```
44                  Z_old_loc[jl+1] = Z_old[(i-1)*width+j];
45                  Qx_old_loc[jl+1] = Qx_old[(i-1)*width+j];
46                  Qy_old_loc[jl+1] = Qy_old[(i-1)*width+j];
47          }
48          if(jl == 0)
49          {
50                  Z_old_loc[(il+1)*ls[1]] = Z_old[i*width+j
                        -1];
51                  Qx_old_loc[(il+1)*ls[1]] = Qx_old[i*width+j
                        -1];
52                  Qy_old_loc[(il+1)*ls[1]] = Qy_old[i*width+j
                        -1];
53          }
54          if(il == get_local_size(0)-1)
55          {
56                  Z_old_loc[(il+2)*ls[1]+jl+1] = Z_old[(i+1)*
                        width+j];
57                  Qx_old_loc[(il+2)*ls[1]+jl+1] = Qx_old[(i+1)
                        *width+j];
58                  Qy_old_loc[(il+2)*ls[1]+jl+1] = Qy_old[(i+1)
                        *width+j];
59          }
60          if(jl == get_local_size(1)-1)
61          {
62                  Z_old_loc[(il+1)*ls[1]+jl+2] = Z_old[i*width
                        +j+1];
63                  Qx_old_loc[(il+1)*ls[1]+jl+2] = Qx_old[i*
                        width+j+1];
64                  Qy_old_loc[(il+1)*ls[1]+jl+2] = Qy_old[i*
                        width+j+1];
65          }
66          il = il+1;
67          jl = jl+1;
68          barrier(CLK_LOCAL_MEM_FENCE);
69          //i calcoli dobbiamo farli solo tra 1 e Nx o Ny e
                indici vanno da 0 ad almeno Nx+1 o Ny+1
70          if(i != 0 && j != 0 && i < Nx+1 && j < width-1)
71          {
72                  /*Corecode*/
73                  Z[i*width+j]=0.25*(Z_old_loc[(il+1)*ls[1]+jl
                        ]+Z_old_loc[(il-1)*ls[1]+jl]+Z_old_loc[il
                        *ls[1]+jl+1]+Z_old_loc[il*ls[1]+jl-1])-dt
                        /(2.0*dx)*(Qx_old_loc[(il+1)*ls[1]+jl]-
                        Qx_old_loc[(il-1)*ls[1]+jl]+Qy_old_loc[il
                        *ls[1]+jl+1]-Qy_old_loc[il*ls[1]+jl-1]);
74                  if(usemass)
75                  {
76                          mass[i*width+j] = Z[i*width+j]*dx*dx
                                ;
77                  }
```

```
78                  Fpx=Qx_old_loc[(il+1)*ls[1]+jl]*Qx_old_loc[(
                        il+1)*ls[1]+jl]/Z_old_loc[(il+1)*ls[1]+jl
                        ]+0.5*9.8*Z_old_loc[(il+1)*ls[1]+jl];
79                  Fmx=Qx_old_loc[(il-1)*ls[1]+jl]*Qx_old_loc[(
                        il-1)*ls[1]+jl]/Z_old_loc[(il-1)*ls[1]+jl
                        ]+0.5*9.8*Z_old_loc[(il-1)*ls[1]+jl];
80                  Gpx=Qx_old_loc[(il+1)*ls[1]+jl]*Qy_old_loc[(
                        il+1)*ls[1]+jl]/Z_old_loc[(il+1)*ls[1]+jl
                        ];
81                  Gmx=Qx_old_loc[(il-1)*ls[1]+jl]*Qy_old_loc[(
                        il-1)*ls[1]+jl]/Z_old_loc[(il-1)*ls[1]+jl
                        ];
82                  Qx[i*width+j]=0.25*(Qx_old_loc[(il+1)*ls[1]+
                        jl]+Qx_old_loc[(il-1)*ls[1]+jl]+
                        Qx_old_loc[il*ls[1]+jl+1]+Qx_old_loc[il*
                        ls[1]+jl-1])-dt/(2*dx)*(Fpx-Fmx+Gpx-Gmx);
83                  Fpy=Qx_old_loc[il*ls[1]+jl+1]*Qy_old_loc[il*
                        ls[1]+jl+1]/Z_old_loc[il*ls[1]+jl+1];
84                  Fmy=Qx_old_loc[il*ls[1]+jl-1]*Qy_old_loc[il*
                        ls[1]+jl-1]/Z_old_loc[il*ls[1]+jl-1];
85                  Gpy=Qy_old_loc[il*ls[1]+jl+1]*Qy_old_loc[il*
                        ls[1]+jl+1]/Z_old_loc[il*ls[1]+jl
                        +1]+0.5*9.8*(Z_old_loc[il*ls[1]+jl+1]);
86          Gmy=Qy_old_loc[il*ls[1]+jl-1]*Qy_old_loc[il*ls[1]+jl
                -1]/Z_old_loc[il*ls[1]+jl-1]+0.5*9.8*(Z_old_loc[
                il*ls[1]+jl-1]);
87                  /*computation of friction term along y*/
88                  u=Qx[i*width+j]/Z[i*width+j];
89                  v=Qy[i*width+j]/Z[i*width+j];
90                  S=(n*n)*v*sqrt(u*u+v*v)/pow(Z[i*width+j
                        ],4.0/3.0);
91                  Qy[i*width+j]=0.25*(Qy_old_loc[(il+1)*ls[1]+
                        jl]+Qy_old_loc[(il-1)*ls[1]+jl]+
                        Qy_old_loc[il*ls[1]+jl+1]+Qy_old_loc[il*
                        ls[1]+jl-1])-dt/(2*dx)*(Fpy-Fmy+Gpy-Gmy);
92          }//end if
93  }
```

Dopo l'iniziale dichiarazione delle variabili e il recupero delle dimensioni globali e locali si convertono (alle righe 33-34) gli indici globali dei work-item nei rispettivi indici locali utilizzando l'operatore resto e di seguito alla dimensione locale degli work-group viene sommato 2, questo valore sarà utilizzato per la larghezza delle matrici da memorizzare nella memoria locale prendendo il ruolo che aveva la variabile *width* nelle precedenti implementazioni. È necessario addizionare due perché le matrici non hanno una corrispondenza biunivoca con il numero di work-item avviati infatti in questa implementazione le matrici locali hanno un bordo (o *padding*) spesso una cella per ospitare le copie dei valori adiacenti. Il kernel non ha alcuna cognizione di quanto siano grandi i dati su cui opera (né globali né locali) poiché questa è decisa nel codice *host* però può

accedere facilmente alle dimensioni *global* e *local* e si è così evitato di gestire esplicitamente con dei parametri la dimensione delle matrici locali.

Nelle righe 39-41 si copiano tutti i valori nelle rispettive celle in memoria locale, ogni work-item copia la propria cella dalla memoria globale in una sua cella locale, non esistono collisioni perché nonostante il valore di posizione (l'indice) della memoria locale sia identico tra i diversi work-item con ciclicità *ls[0]* o *ls[1]* quando questo accade gli work-item si trovano in work-group diversi e il riferimento quindi è a memorie locali differenti (si ricorda che è una per ogni work-group).

Nelle righe 42-65 si preparano i bordi delle diverse matrici locali, nel caso ci si trovi in una cella sul bordo viene copiato il valore adiacente (che altrimenti sarebbe appunto copiato solo nella rispettiva memoria locale dalle righe di codice precedenti). L'accesso ai valori adiacenti nella memoria globale è chiaro con l'aggiunta o la sottrazione di 1 agli indici i e j, in memoria locale è poi necessario sommare uno a entrambi gli indici corrispondenti poiché le memorie locali contengono i bordi in entrambe le dimensioni (questo risulta a volte in un indice 0 che non viene riportato). Le righe 66-67 aggiungono uno agli indici locali da utilizzare poiché utilizzando il resto si ottengono indici da *0* a *ls[0]-1* (o *ls[1]-1*) per un totale di celle di $ls[0]ls[1]$ ma in seguito sono necessari valori da 1 a *ls[0]* (o *ls[1]*) in modo da riutilizzare gli indici degli work-item come indici per le celle (la memoria locale è infatti grande $(ls[0]+2)(ls[1]+2)$ e i calcoli non devono essere effettuati per riempire le celle sul bordo che sono state incluse solo per evitare di recuperare i dati dalla memoria globale, il numero totale di celle è sempre $ls[0]ls[1]$). Subito dopo è necessaria una barriera fino a che le copie in memoria locale non sono state completate che assicura che nessun work-item in un singolo work-group proseguirà la computazione oltre la barriera fino a che gli altri non avranno finito la porzione di codice precedente la barriera, altrimenti nelle celle potrebbe trovare valori indefiniti. Si fa notare che è sufficiente una barriera locale e non globale infatti la memoria globale è acceduta solo in lettura e non ci sono altri work-item o codice host in esecuzione che possano modificarla durante quegli accessi inoltre ogni work-item accede indipendentemente ad una sola cella. La condizione dell'istruzione condizionale alla riga 70 si riferisce invece al numero globale di work-item, poiché a contrario delle versioni precedenti il numero totale degli work-item è superiore in quanto esistono almeno anche work-item associati al bordo della griglia totale che devono copiare il rispettivo valore in memoria locale (nelle versioni precedenti a questi item non erano associati calcoli o copie e il valore veniva recuperato dalla memoria globale). Oltre a questo possono esserci altri work-item in eccedenza infatti non vengono più usati Nx e Ny come dimensioni globali ma il primo multiplo della dimensione locale superiore rispettivamente a *Nx+2* e *Ny+2* per le due dimensioni; i calcoli devono poi essere effettuati rispettivamente nell'intervallo [1: Nx] e [1:Ny].

Infine si calcolano i nuovi valori che vengono riposti immediatamente nella memoria globale.

In figura è mostrato un esempio di memoria locale utilizzata (*Z_old_loc* di dimensione 18x18, come si otterrebbe eseguendo l'implementazione riportata su una GPU Radeon serie 5800 che ha dimensione massima locale 256 sia in totale

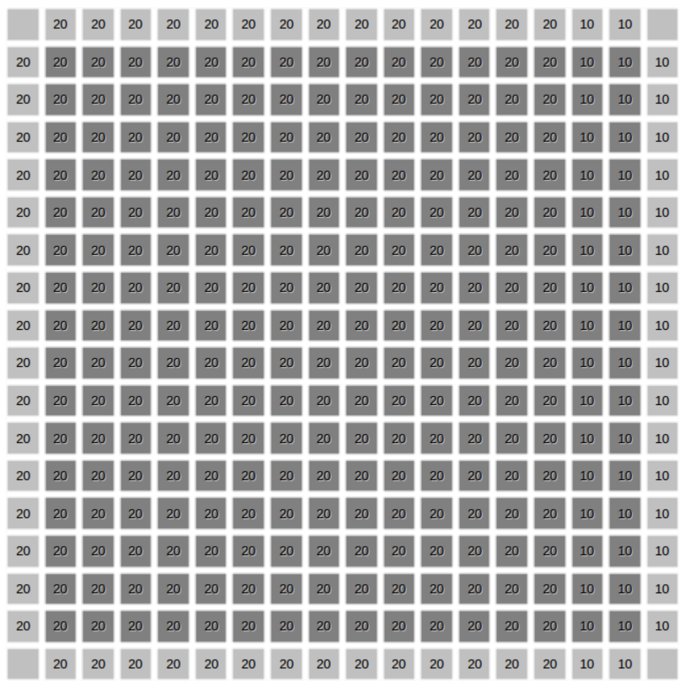


Figura 4.1: Esempio di memoria locale

sia per ogni dimensione), le celle grigio scuro sono quelle direttamente associate agli work-item del singolo work-group che effettueranno i calcoli, le celle grigio chiaro sono quelle il cui valore è stato recuparato dagli work-item associati ai bordi della zona più scura. I quattro angoli non contengono valori poiché non vengono riempiti da nessun work-item, non è necessario poiché ogni work-item che effettua i calcoli avrà bisogno soltanto delle quattro celle direttamente adiacenti per completare le operazioni. Le celle negli angoli della zona scura copiano due valori dalla zona chiara senza che sia necessario gestirlo esplicitamente, infatti sono sia nel bordo verticale sia nel bordo orizzontale, quello adiacente in orizzontale e quello adiacente in verticale.

Un'alternativa all'approccio qui descritto è quello di dimensionare il numero di work-item già in accordo con il numero di celle, incluso il bordo, della memoria locale di ogni work-group.

4.5.3 Riduzione dei trasferimenti di memoria

Con l'uso della memoria locale si riducono i continui accessi alla memoria globale con un solo trasferimento iniziale e poi l'accesso locale ai dati, tuttavia questo accorgimento non è sufficiente per migliorare significativamente le prestazioni rispetto all'esecuzione su CPU. La memoria locale può essere però sfruttata per evitare il continuo scambio tra i buffer che rappresentano i valori al passo precedente (in realtà solo una copia, poiché i buffer del passo corrente saranno riscritti è necessario soltanto copiare i buffer correnti nel rispettivi buffer "old"). Utilizzando la memoria locale è possibile evitare di mantenere variabili sia per il passo corrente (per memorizzare i risultati dei calcoli) sia per il passo precedente (i dati che devono essere utilizzati dai calcoli) infatti dopo la copia iniziale i dati sono disponibili interamente nelle diverse memorie locali (una per ogni work.group) e si possono usare i buffer di input per memorizzare direttamente i risultati senza corrompere i dati da utilizzare nello stesso passo, in questo modo vengono ridotti i trasferimenti tra la memoria dell'host (la RAM) e quella della GPU. Il codice host è del tutto equivalente a quello della prima versione utilizzante OpenCL riportata in 4.2 con le seguenti differenze

- l'invocazione del kernel ha meno parametri (dovendo operare solo sulle variabili "_old")
- non sono più necessarie le chiamate a funzioni che operano la copia tra diversi buffer (infatti come spiegato lo stesso buffer è usato sia per memorizzare i dati del passo precedente sia per memorizzare i risultati e alla fine di ogni esecuzione degli work-item questo contiene automaticamente i dati per il passo successivo)

4.5.4 Unione dei metodi precedenti

Si sono introdotti nella stessa versione sia la copia parallela di righe e colonne (eliminando la necessità di risincronizzare i buffer OpenCL dopo la copia) sia

la memoria locale (con la riduzione dei trasferimenti di memoria esposta nella sezione precedente tramite eliminazione della necessità della copia dei valori correnti nei buffer che al passo successivo saranno considerati i valori del passo precedente), in questo modo il ciclo di calcolo opera esclusivamente tramite kernel OpenCL, recuperando i dati esclusivamente nei passi decisi per l'output su file (o eventualmente a video). Questa versione elimina quindi sia la necessità di copiare gli stessi dati da un buffer all'altro (come spiegato nella sezione precedente) sia quello di recuperare i dati alla fine del ciclo (come spiegato in 4.2). Nelle versioni sequenziali non era necessario copiare i valori correnti nelle corrispettive variabili relative al passo precedente poiché questo era effettuato tramite uno scambio tra i puntatori, quindi poco costoso. Nelle altre versioni OpenCL invece si procede alla copia dei valori correnti nelle variabili "old" tuttavia si sarebbe potuto risparmiare la copia anche in quel caso, e senza utilizzare la memoria locale. La tecnica consiste nell'utilizzo per ogni ciclo due chiamate per avviare gli work-item effettuando la seconda con le variabili invertite in modo che i risultati della chiamata precedente diventino le variabili "old" di quella succesiva e che le variabili "old" della chiamata precedente possano accogliere i nuovi risultati poiché i dati in esse contenuti non servono più per i calcoli successivi (in questo caso si dovrebbe effettuare un numero di cicli pari alla metà delle iterazioni desiderate). Un esempio di questo approccio si trova nell'esempio di simulazione dell'interazione gravitazionale tra n corpi riportato in[BDT, 2011] Come spiegato in precedenza con la memoria locale si possono risparmiare copie utilizzando un solo tipo di variabili senza distinguere tra valori vecchi e correnti, l'approccio della doppia chiamata è però utile nel caso non si possa o non si voglia utilizzare la memoria locale (nel caso ad esempio essa degradi le prestazioni o sia troppo complessa da gestire per il problema in esame) infatti effettuando i calcoli su GPU le copie sono costose perché richiedono una comunicazione con la memoria *host*. Si è costruita anche una versione che non distingue tra variabili *old* e *new* utilizzando la memoria locale ma senza utilizzare la copia parallela di righe e colonne. Le versioni descritte in questa sezione restituiscono risultati coerenti con le altre implementazioni ma, anche se identici per dimensioni piccole della griglia (ad esempio 60x40), si sono riscontrate differenze marginali all'aumentare delle dimensioni, probabilmente dovute alla mancata comunicazione con l'*host*.

4.5.5 Tecniche Single Instruction Multiple Data (SIMD)

Per i calcoli parallelizzabili è spesso efficiente utilizzare un approccio SIMD (come quello utilizzato dalle istruzioni SSE delle CPU di tipo x86 e x86-64 o dalle istruzioni Altivec dell'architettura PowerPC o dalle recenti Advanced Vector Extension supportate da Intel con i processori "Sandy brige" e da AMD nei processori "Bulldozer") che rappresenta i dati come vettori (di solito 4 elementi) su cui vengono effettuate parallelamente operazioni con hardware specifico. Anche le GPU possono utilizzare questo approccio ai dati (le GPU della serie 5800 di AMD possono effettuare operazioni in virgola mobile su vettori di 4 elementi). Un esempio banale è la somma di due vettori che viene velocizzata se invece di

effettuare un ciclo che effettua l'operazione per una singola cella ad ogni passo il ciclo riesce a sommare ad ogni passo i valori di 4 caselle ai valori delle caselle corrispondenti nell'altro vettore. I quattro campi di ogni vettore di 4 elementi possono essere recuperati accedendo con la notazione punto ai campi x, y, z o w di un particolare valore (una variabile o una cella di un array).

Anche se il problema non sembrava adatto a sfruttare in maniera adeguata un approccio di questo tipo si è deciso di rappresentare i dati come vettori di *float4* e costruire un kernel che calcolasse in maniera corretta i risultati operando sui dati in questo formato. La difficoltà è stata l'approccio ai bordi della matrice dei risultati, poiché la singola cella è rappresenata da 4 elementi e non devono essere utilizzati i valori del bordo della matrice, si è quindi deciso di inserire un *padding* di 3 elementi a destra e 3 elementi a sinistra in modo che i valori utili siano contenuti rispettivamente nell'ultimo campo (w) e nel primo (x) delle celle ai bordi. Per la dimensione verticale (il numero di righe, cioè *Nx+2*) questo non è stato necessario perché le matrici sono rappresentate come vettori che è stato scelto di interpretate come righe affiancate quindi il fatto che la singola cella contenga ora quattro elementi non influisce su questo, rimangono comunque da scartare la prima e l'ultima riga. Lo stesso vale per l'accesso alle celle adiacenti, per quelle in una riga diversa non cambia nulla rispetto alle altre versioni mentre per le celle adiacenti sulla riga bisogna scegliere il campo corretto (che può essere x, y, z o w) nella stessa cella o in una cella adiacente in modo dipendente dalla posizione del valore che deve essere calcolato. Questo approccio non ha prodotto significativi aumenti di prestazione rispetto alle altre versioni utilizzanti OpenCL.

Un approccio di questo tipo invece è molto utile nel caso di copie, interpretando i dati come un vettore di float4 invece di float1 le copie possono essere ridotte a un quarto degli elementi e, poiché la serie Radeon HD 5800 supporta trasferimenti di 128 bit pari alla dimensione di un dato float4, rappresentando i dati in questo modo si sfrutta al massimo la banda disponibile per ogni copia. È stata realizzata una versione che sfrutta questo approccio per la copia di righe, non si è utilizzata per la copia di colonne poiché le matrici sono rappresentate come righe affiancate, non è stato quindi possibile sfruttare questo approccio per la copia di colonne. Per utilizzare questo approccio è necessario che la lunghezza delle righe sia un multiplo di quattro perciò la variabile width non può essere inizializzata a Ny+2 e il codice host ha le seguenti differenze:

- inizializzazione di `width` al primo multiplo di 4 maggiore di Ny+2
- le allocazioni dei dati su cui operare invece di utilizzare dimensione `(Nx+2)*(Ny+2)` utilizzano dimensione `(Nx+2)*width` (questo si potrebbe utilizzare anche in tutte le altre versioni poiché a `width` è assegnato il valore Ny+2 come quello utilizzato nelle allocazioni però le implementazioni sono tutte derivate dall'implementazione seriale che non necessitava di `width` che è stato introdotta nelle implementazioni OpenCL per il passaggio di parametri al kernel e poi nella versione con vettori puri che originariamente utilizzava esplicitamente Ny+2, questa è l'unica versione che ha richiesto una dimensione diversa delle allocazioni)

- le copie di righe sono invocate con un numero di work-item pari a un quarto del parametro `size2` (che corrisponde al parametro effettivo, o attuale, `width`)

Le inizializzazioni sono invece identiche e le matrici sono riempite da 0 a Nx+1 in una dimensione e da 0 a Ny+1 nell'altra (in questo caso è stato utile nella versione precedente riferirsi direttamente a Ny invece che a width).

Nel kernel invece la funzione *calculate* è identica a quella della corrispondente versione non float4 (che è a sua volta identica a quella della prima versione OpenCL), infatti può interpretare la matrice come una normale matrice di float e le celle in più delle righe non vengono mai lette e non interferiscono con i calcoli (come nelle prima parallelizzazione riportata gli item sono numerati da 0 a Nx-1 in una dimensione e da 0 a Ny-1 nell'altra e gli indici vengono poi slittati di 1 associando gli item rispettivamente agli indici da 1 a Nx e da 1 a Ny e poiché ogni cella accede alle quattro celle adiacenti si utilizzano le celle da (0,0) a (Nx+1,Ny+1)); le differenze sono unicamente nelle funzioni *copyrow* che indicano le matrici su cui operare di tipo puntatore a float4 invece di puntatore a float semplice.

4.6 Altre considerazioni

Si riportano in questa sezione una serie di considerazioni della documentazione AMD per la realizzazione di codice efficiente (in particolare su GPU della serie Cypress di AMD). GPU e CPU hanno entrambe prestazioni elevate ma hanno ognuna aspetti specifici: la GPU ha un maggiore throughput e può nascondere la latenza, la CPU disponde invece di bassa latenza, cache e tempo di lancio minore.

Uno dei vantaggi principali di OpenCL è che ogni work-item può accedere a indirizzi arbitrari di memoria, questo approccio è più flessibile rispetto all'approccio delle istruzioni SSE su CPU.

I calcoli in singola precisione sono molto più efficienti di quelli in doppia precisione, un'operazione MAD (multiply and add) ha in singola precisione un throughput 5 volte maggiore rispetto alla singola precisione (la doppia precisione inoltre è supportata solo sui dispositivi Cypress e seguenti, cioè dalla serie 5800).

Uno dei parametri da considerare è la dimensione della memoria locale, è bene evitare dimensioni piccole poiché i trasferimenti verso la GPU sono costosi, infatti nelle implementazioni utilizzanti la memoria locale riportate in 4.5.2 si è scelta la massima dimensione consentita dal numero di work-item, con dimensioni più piccole gli accessi duplici in memoria globale (da parte di item di diversi work-group per riempire i bordi delle proprie memorie locali) sarebbero moltiplicati. Questo implica anche che spesso con la GPU è più efficiente ricalcolare i dati piuttosto che salvarli e recuperarli in un secondo momento dalla memoria globale.

Come spiegato in 3.6.1 gli stream core della GPU sono costituiti da processori VLIW, sono flessibili ed efficienti in singola precisione anche senza l'utilizzo esplicito di codice float4, il codice float4 genera però codice ancora più efficiente.

La latenza *read-after-write* della maggior parte delle operazioni aritmetiche è di 8 cicli. Per la maggior parte delle GPU AMD ogni *compute unit* può eseguire 16 istruzioni VLIW in un singolo ciclo. Un gruppo di work-item di 64 elementi denominato *wave-front* viene quindi eseguito per un quarto in un ciclo. Per questo quindi è meglio non utilizzare work-group di dimensione inferiore a 64 e per lo stesso motivo il numero degli work-item in esso è consigliato essere un multiplo intero di 64. Work-group di dimensione elevata sono consigliati finché restano in numero sufficiente da occupare ogni *compute unit* con 1 o 2 work-group (in una simulazione 100x100 con memoria locale utilizzando la dimensione massima ci sono 40 work-group, ognuno di 256 elementi, sufficienti a occupare le 14 *compute units*). Nel caso utilizzando una dimensione grande gli work-group fossero in numero così ridotto da lasciare inoperative alcune unità sarebbe consigliato diminuire la dimensione per aumentare il numero di work-group in modo che siano distribuiti su più unità. La serie Radeon 5800 supporta un numero diverso di wave-front per ogni work-group dipendente dal numero di registri utilizzati da ogni item: più sono i registri utilizzati da un item meno wavefront possono essere utilizzati. Questo si può sfruttare anche al contrario, con un basso numero di work-group è possibile specificare a tempo di compilazione il numero di work-group poiché il compilatore OpenCL AMD dei kernel utilizza un limite massimo di registri per un singolo work-item presumendo la dimensione massima di di uno work-group (cioè, per la serie Radeon 5800, 62 registri per uno work-group di 256 work-item cioè 4 wavefront) generando *spill code* se necessario per non eccedere il numero massimo di registri consentito, utilizzando però work-group più piccoli e indicandolo a tempo di compilazione sono disponibili più registri per ogni work-item. L'ATI Stream Profiler indica il numero di registri utilizzato e permette di controllare che non sia stato generato *spill code* durante la compilazione.

Evitare i conflitti di memoria in OpenCL è una pratica importante infatti, poiché le GPU dispongono di più banchi che possono essere indirizzati in una volta sola dal bus della memoria globale, indirizzare contemporaneamente banchi dversi aumenta la prestazioni, per le GPU il problema non è il numero degli accessi ma il tempo di risposta che aumenta nel caso le richieste debbano essere soddisfatte in momenti successivi e non contemporaneamente.

Per accedere a un'indirizzo fisso in memoria globale il cui valore debba essere recuperato è più efficiente farlo copiare in memoria locale da un solo work-item in questo modo:

```
if (get_local_id(0) == 0)
{
        local = input[3];
}
barrier(CLK_LOCAL_MEM_FENCE);
temp = local
```

piuttosto che utilizzarlo in questo:

```
temp = input[3]
```

infatti nel secondo caso ci sarebbe un conflitto di accesso tra *tutti* gli work-item in esecuzione. Sempre a proposito dell'accesso alla memoria globale per aumentare le prestazioni si possono eliminare i conflitti tra canali di memoria facendo in modo che ogni work-item dei 64 che compongono un wave-front legga da un'indirizzo di memoria diverso in una regione di 64 parole di memoria. Calcolando un indirizzo come `y*width+x` e leggendo incrementado y solo un canale di memoria è utilizzato poichè *width* è probabilmente un multiplo di 256. Se tutti gli work-item in un work-group accedono a indirizzi consecutivi di memoria essi utilizzano un solo canale. Questo vale anche nel caso non vi siano cicli all'interno del kernel come una copia globale di un oggetto bidimensionale (tuttavia per le copie di buffer OpenCL sono disponibili funzioni OpenCL apposite e non è necessario accedere tramite gli indici a tutti gli elementi in memoria). In casi in cui sia necessario l'accesso con due indici si può fare in modo che gli work-item di un wave-front accedano a indirizzi consecutivi utilizzando una dimensione unitaria (ad esempio per dati *float* 1x64 o 64x1, la direzione dipende da come è stata strutturata la matrice, altrimenti un wave-front utilizzerebbe più canali e ci sarebbero conflitti con altri wave-front). Anche se non è consigliato utilizzare work-group che non siano multipli di 64 questo è meno importante dell'evitare i conflitti di memoria e si possono quindi utilizzare dimensioni non multiple. Per il caso riportato in questa tesi i conflitti possono riguardare le copie da globale a locale e le copie di righe e colonne (tuttavia in questo caso evitando problemi per le colonne si ottengono problemi per le righe e viceversa). Le stesse considerazioni si possono applicare ai conflitti tra banchi di memoria, infatti al variare di numero di canali quello che è un conflitto di canale su una macchina può essere di un banco in un'altra tuttavia entrambi riguardano potenze elevate di due. Poiché un accesso alla memoria globale richiede una linea larga quanto il bus un conflitto in un work-group è simile ad un accesso non coalescizzato comunque non tutti gli accessi non coalescizzati sono conflitti di memoria.

Per le GPU della serie 5800 gli work-item utilizzano due modi diversi di recuperare la dati dalla memoria globale: FastPath e CompletePath molto più lento (per le operazioni atomiche e dati rappresentati da meno di 32 bit) quindi se non sono strettamente necessarie si possono aumentare le prestazioni eliminando queste caratteritiche (può anche essere utile verificare con il profiling che il CompletePath non sia stato scelto dalla compilazione dei kernel).

Un altro modo di aumentare le prestazioni del codice su GPU è sfruttare la coalescenza delle letture e delle scritture dalla memoria globale quando sono presenti cicli all'interno del kernel. Se ad esempio diversi work-item devono accedere ad uno stesso vettore e non sono in numero equivalente agli elementi del vettore ma inferiori essi dovranno accedere con un ciclo a più elementi del vettore; se diversi item iniziano accedendo a elementi adiacenti ed ogni item utilizza un certo passo (*stride*) equivalente al numero di item per attraversare il vettore, invece di accedere in ordine ad elementi contigui, i diversi item eseguiti in parallelo in ogni iterazione del ciclo accedono allo stesso blocco di memoria richiedendo (sfruttando l'ampiezza del bus di memoria, di 256 bit per la serie Cypress) quindi un numero inferiore di blocchi di memoria per l'intera elaborazione. Per le CPU l'approccio migliore è l'opposto, associare ad ogni thread

un blocco di elementi del vettore adiacenti da scandire in ordine per sfruttare al meglio la cache e il prefetching del processore. La coalescenza delle scritture del tutto analoga è supportata dalla serie ATI Radeon HD 5800 ma su questo dispositivo è meno importante rispetto alle altre ottimizzazioni, è invece più importante sui prodotti CUDA. Per la memoria LDS non è necessaria invece la coalescenza.

Anche i conflitti tra banchi di memoria del LDS riducono le prestazioni (è possibile individuarli effettuando un profiling del codice), la memoria LDS può infatti in un ciclo servire una sola richiesta per ogni banco di memoria e nel caso ci sia un conflitto l'intero wave-front rimane in attesa fino a che tutti gli accessi alla memoria LDS non sono completati. Il conflitto vale però solo nel caso il banco di memoria sia lo stesso ma l'indirizzo diverso altrimenti la memoria LDS può servire tutti gli work-item dallo stesso indirizzo.

Usare un work-flow complesso riduce molto l'efficienza dell'esecuzione di un kernel OpenCL su GPU infatti tutti gli work-item eseguono comunque le istruzioni che non sono nel loro flusso ideale di esecuzione basato sulle condizioni perciò un *if...else* spende comunque il tempo di entrambi i blocchi infatti tutti gli work-item eseguono prima un blocco e successivamente l'altro, allo stesso modo un ciclo ha la complessità del caso peggiore. Per questi motivi sono consigliati l'operatore ternario ? : in modo predicativo al posto dell'istruzione condizionale *if...else* per gli assegnamenti con due alternative dipendenti da una condizione, ad esempio `int C = (A>B) ? 1:-1;`, e il *loop unrolling* che viene comunque effettuato in modo semplice dal compilatore OpenCL ma può essere anche eseguito manualmente (esempi tipici in cui si può utilizzare sono la moltiplicazione tra matrici è la convoluzione di immagine di cui AMD fornisce l'esempio come riportato in).

L'eliminazione di sincronizzazioni non necessarie migliora le prestazioni, ad esempio un work-group di dimensione uguale o inferiore a uno wave-front è già sincronizzato internamente (si può considerare un'unità atomica di esecuzione), tuttavia fare queste considerazioni diminuisce la portabilità.

Una modifica che può migliorare in modo marginale le prestazioni ma che può essere provato se gli altri metodi non hanno dato il risultato atteso o non sono applicabili è l'accesso allineato ai dati in memoria globale ma questo sulle GPU di AMD provoca variazione minime (nella documentazione AMD è consigliato come ultimo tentativo per migliorare le prestazioni).

I kernel OpenCL possono essere eseguiti anche su più dispositivi, in questo caso è necessario e utilizzare *clFlush* su ogni *command queue* prima di *clFinish* (utilizzato per attendere la fine delle esecuzioni nella *command queue*).

Si può notare che la maggior parte delle tecniche qui riportate non riguardano il caso riportato in questa tesi perché utilizza un approccio di tipo *stencil* a cinque punti (ogni work-item ha bisogno solo di una cella e della quattro adiacenti) che non richiede cicli all'interno di uno work-item.

4.6.1 Riduzione nel numero di work-item

Come spiegato in 3.6 e 4.6 le GPU hanno molti elementi in grado di lavorare in parallelo (chiamati *stream cores* sulle GPU AMD) eseguendo operazioni semplici ma che offrono prestazioni scarse con controlli di flusso complessi, al contrario le CPU hanno pochi elementi in grado di eseguire efficientemente codici complessi. L'approccio di utilizzare uno work-item per ogni cella sfrutta al meglio la GPU ma sulla CPU richiede l'esecuzione di molti threads su ogni core. A causa di questo si è costruita una versione che utilizza un numero di work-item pari al numero di compute units (per la CPU è il numero di core, compresi quelli logici) dividendo il dominio in sezioni o "fette" e ogni item esegue in modo sequenziale con due *for* innestati i calcoli associati ad ogni cella a lui associata mentre gli altri item eseguono parallelamente i loro calcoli. Come già spiegato per la GPU i flussi complessi sono da evitare e tali sono anche i due *for* innestati tuttavia i test per questa versione, pensata per la CPU, sono stati eseguiti anche per GPU per valutarne la differenza di prestazione. Come già indicato un metodo per migliorare le performance dei cicli *for* per l'esecuzione su GPU può essere l'utilizzo di un unrolling esplicito tuttavia in questo caso i cicli non sono necessari poiché è possibile utilizzare l'approccio di associare un item ad ogni cella, sfruttando al meglio le molte unità parallele della GPU. Il codice è stato eseguito in due versioni, una non indica dimensioni locali, l'altra invece indica una dimensione locale unitaria come l'implementazione Intel descritta in 4.7 in modo da forzare l'esecuzione di ogni item su un diverso core. È stato possibile realizzare questa versione perché nonostante l'approccio più semplice per parallelizzare sia associare celle e work-item questo non è richiesto dalla specifica, inoltre le dimensioni di uno work-group non devono necessariamente coincidere con il numero di celle della memoria locale (la dimensione di un work-group e della memoria locale sono due cose diverse nonostante sia comodo utilizzare un approccio che metta in relazione le due dimensioni generando così anche codice più comprensibile).

4.7 Implementazione Intel

Intel fornisce nel suo Intel OpenCL SDK un esempio di implementazione di schema delle acque basse interattivo eseguito nel runtime OpenCL e con output grafico tramite DirectX (la demo è infatti disponibile solo per Windows). Si è adattato facilmente il codice all'esecuzione su GPU (nella stessa configurazione utilizzata per i test al capitolo successivo con la GPU Radeon ma in Windows 7 Professional 64-bit e Microsoft Visual Studio 2008 e con i driver Catalyst 11.7 (8.872-110707b-122569C-ATI) e ATI Stream SDK v2 tuttavia il calo di prestazioni è stato notevole anche se si è ridotto aumentando la scelta della dimensione globale e la dimensione degli degli work-group (che comunque in tale implementazione non sfruttano la memoria locale, quindi gli work-item sono divisi in work-group ma non condividono memoria). L'applicazione originale utilizza infatti un numero di work-item pari al numero di compute unit

Figura 4.2: Simulazione Intel Shallow Water originale eseguita su CPU

rilevate (su GPU invece una singola compute unit può eseguire molti work-item, al contrario dei core della CPU che possono eseguire un solo thread ognuno) e dimensione locale pari a 1, il codice quindi forza l'esecuzione di un thread su ogni core (con dimensione locale pari a uno un core non può eseguire più di un item e poiché gli item sono in numero pari a a quello di core ogni core ha un item da eseguire). Si sono aumentate leggermente le prestazioni aumentando la dimensione globale a 3072 e quella locale a 16 (il risultato è riportato nella terza figura). Il codice dell'esempio di Intel non utilizza accorgimenti particolari e, nonostante le elevate performance disponibili su CPU (riportate nella prima figura, la dicitura HARDWARE: ATI Radeon HD 5800 Series si riferisce all'hardware per l'output a video e non a quello utilizzato per l'esecuzione degli work-item) la GPU è nettamente inferiore quindi tale codice non ha fornito spunti per migliorare le prestazioni su GPU della versione prodotta per questa tesi.

4.8 Considerazioni sull'aggiunta di grafica

OpenCL permette di utilizzare direttamente in OpenGL i dati resituiti dalla GPU evitando così la comunicazione con la CPU, per applicazioni che non han-

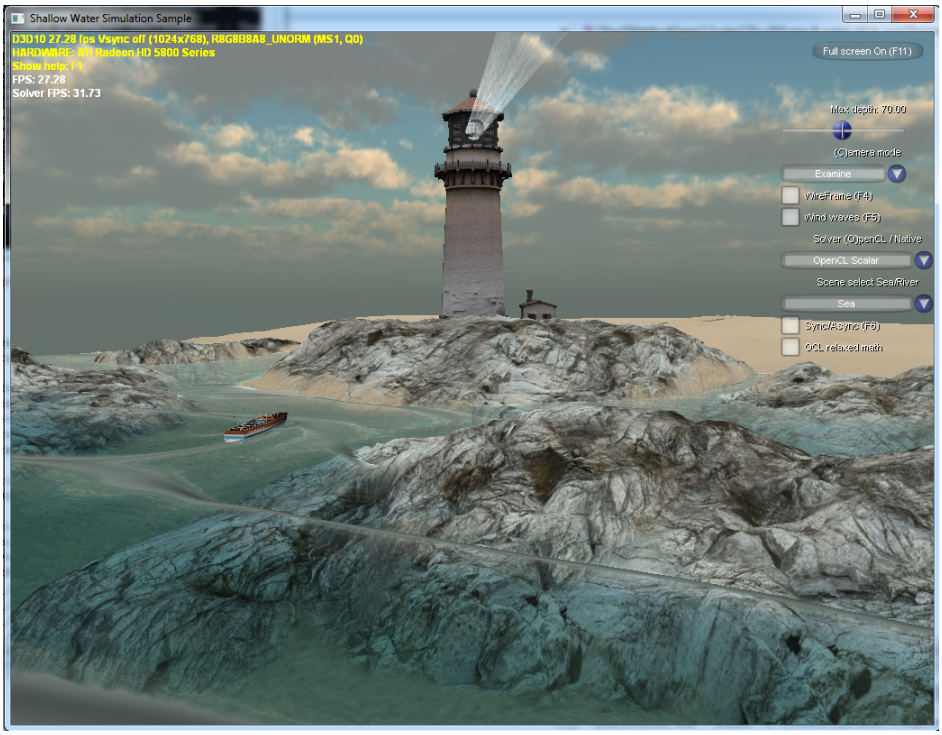


Figura 4.3: Simulazione Intel Shallow Water eseguita su GPU

Figura 4.4: Seconda simulazione Intel Shallow Water eseguita su GPU

no bisogno di output su memorie di massa tramite file o che comunque non richiedano il passaggio dei dati alla CPU per l'output, ad esempio applicazioni interattive che richiedono un'uscita esclusivamente grafica, questo è vantaggioso nel caso di esecuzione dei calcoli su GPU. Si fa notare che nel caso di esecuzione degli work-item OpenCL su CPU il passaggio dei dati calcolati dalla CPU alla GPU è necessario (ma in quella sola direzione) invece nel caso non si sfrutti l'interoparibilità tra OpenCL e OpenGL nel caso di esecuzione su GPU sarebbe necessario prima un trasferimento dei dati dalla GPU alla CPU e di seguito un passaggio dei dati dalla CPU alla GPU con la richiesta di esecuzione grafica.

Utilizzando OpenCL sia su CPU sia su GPU emerge una prima decisione:

- decidere se fermare completamente la simulazione per produrre l'otput a ogni passo che lo richieda e riprendere l'esecuzione successivamente oppure inserire in un *buffer* di tipo produttore/consumatore più risultati attendendo un determinato momento, come ad esempio il riempimento, per attingere ad esso e mostrare l'output (questo problema si presenta anche nel caso si eseguano gli item su CPU e non è esclusivo dell'esecuzione su GPU)

I problemi principali esclusivi dell'esecuzione su GPU sono due:

- la potenza della GPU sarà a questo punto destinata sia al calcolo della simulazione sia al calcolo dell'output grafico
- è necessario gestire al meglio la sincronizzazione tra i due compiti della GPU decidendo se disaccoppiare completamente i due compiti (ad ogni *frame* calcolato, cioè ogni volta che si attinge dal buffer per l'otput, fermare totalmente i calcoli e produrre l'output) oppure fare in modo che la GPU possa eseguire i calcoli anche mentre è impegnata nell'output grafico

Capitolo 5

Risultati dei test

In questo capitolo dopo aver descritto le caratteristiche hardware e software delle piattaforme e del caso di test utilizzati si presenteranno i tempi di esecuzione delle implementazioni in virgola mobile prodotte sia in doppia sia in singola precisione e si riassumeranno infine i risultati salienti.

5.1 Piattaforme di esecuzione e caso di test

La piattaforma principale per lo sviluppo e e i test su CPU Intel e GPU ATI è stato un personal computer dotato di processore Core i7 930 (2.8 GHz, 4 core e 8 thread hardware), ATI Radeon HD 5830 ("Cypress LE") con 1GB di memoria video (scheda ASUS EAH5830 DirectCU) e driver in versione Catalyst 11.11, 4GB RAM (2 moduli Kingston da 2 GB a 1333 Mhz in dual channel forzati a operare alla frequenza massima nonostante il processore supporti ufficialmente solo 1066 MHz), chipset Intel x58 Express, scheda madre ASUS P6T SE, sistema operativo Ubuntu 10.04 - Lucid Lynx (64 bit) basato su GNU/Linux (con attivata l'opzione "normali" per gli effetti visivi infatti la GPU utilizzata è la stessa utilizzata per il normale output grafico) e AMD APP (Accelerated Parallel Processing) SDK 2.5 a 64 bit. L'AMD APP SDK sostituisce il precedente ATI Stream SDK (si è verificata la compatibilità del codice anche con la versione 2.2 di tale SDK e del rispettivo runtime OpenCL). Per le prove sia su CPU sia su GPU si è forzato il processore alla frequenza massima di 2,8GHz disabilitando l'Ehnanced Intel SpeedStep poiché, sia per le prove CPU sia per le prove su GPU si è riscontrato che il processore era lasciato spesso alla frequenza minima di 1,6 GHz tuttavia si è riscontrato un leggero miglioramento di prestazioni (più significativo nel caso della sola CPU), questo ha comportato anche la disattivazione dell'Intel Turbo Boost. Per le prove su 4 core o meno si è disabilitato l'Intel Simultaneous Multithreading in modo da forzare l'esecuzione sui soli core reali (tranne dove indicato diversamente), per le prove su GPU si sono mantenuti attivi tutti gli 8 core. È possibile forzare l'esecuzione di un programma OpenCL su un preciso numero di *compute unit* impostando

la variabile d'ambiente `export CPU_MAX_COMPUTE_UNITS` al numero desiderato, ad esempio `export CPU_MAX_COMPUTE_UNITS=4`, nei test riportati è comunque stata utilizzata la scelta da BIOS del numero di core attivi per forzare l'esecuzione sugli stessi core durante l'intera esecuzione del programma. Si è utilizzato invece il server Veio del Dipartimento di Matematica del Politecnico di Milano per i test su GPU NVIDIA Tesla C1060 e processore Intel Xeon, è dotato infatti anche di quattro processori quad-core Xeon 5520 a 2,27 GHz e di un totale di 12 GB di RAM, rifersi a [Politecnico di Milano, 2011] per una descrizione dettagliata dell'hardware. Si è utilizzato poi il computer Iperione del Dipartimento di Matematica che dispone di un processore quad-core Phenom X4 9950 a 2,609 GHz, 4GB di RAM e una GPU GTS450 con 1 GB di memoria video.

La GPU Radeon HD 5830 è realizzata secondo l'architettura a 3 livelli della serie Cypress in modo simile a quanto esposto per la Radeon HD 5870 infatti dispone di 1120 *stream processors* o *processing elements*, raggruppati a loro volta in 224 *stream cores*, a loro volta raggruppati in 14 *compute unit*; questa GPU contiene quindi 14 unità di elaborazione, ognuna delle quali contiene 16 *core* ognuno dei quali contiene 5 *processing element*.

La GPU Tesla C1060 è costituita da 30 computation unit, ognuna comprendente 8 *cores* per un totale di 240.

La GPU GTS450 dispone di 4 *compute unit* riconosciute da OpenCL e un totale di 192 *CUDA cores* attivi (corrispondenti agli *stream cores* AMD), ogni unità di elaborazione contiene quindi 48 CUDA core.

Nel caso di esecuzione su CPU le *compute unit* OpenCL corrispondono al numero di core, compresi quelli solo logici.

Il caso di test utilizzato, tranne ove diversamente specificato, è stato il caso di rottura di una diga ideale su una griglia 1000x1000 con discretizzazione spaziale pari a 0.5 m, un livello di 20 m da una parte (prima di 100 m, quindi per 200 celle poiché la discretizzazione è pari a 0.5 m) e un livello di 10 m dall'altra, fondo piatto, tempo finale pari a 20 secondi e *plotstep* pari a 10 (tuttavia l'output non è stato utilizzato per i test di prestazione ma solo per verificare la consistenza tra le diverse versioni utilizzando l'applicazione Meld).

Per i test si è utilizzata una versione del codice che tramite direttive di preprocessore, utilizzando quindi una compilazione condizionale, esclude l'interattività descritta nel capitolo precedente per eliminare la variabilità dovuta al tempo di reazione e digitazione, escludendo l'interattività non è possibile scegliere la discretizzazione temporale che è forzata a quella precalcolata.

Con lo stesso metodo si sono esclusi tutti gli output su file per eliminare l'influenza dell'output su disco poiché il tempo di questa operazione è significativamente più elevato di quello necessario per i calcoli, in particolare con la piattaforma comprendente il processore Core i7 e la GPU AMD, il sever Veio dispone invece di un output su file molto più veloce (200 MB/s [Politecnico di Milano, 2011]). L'accesso alla memoria di massa rimane limitato ai parametri della simulazione, alla scelta di esecuzione su CPU o GPU, all'indicazione di tenere traccia della massa del sistema simulato e al caricamento del kernel per la compilazione.

I tempi misurati si riferiscono all'eseguibile ottenuto, il tempo si riferisce così all'intera preparazione e non solo alla parte puramente di calcolo, si ricorda che l'esecuzione comprende anche l'inizializzazione delle variabili e la compilazione del kernel, si è fatto questo sia per semplicità sia per valutare l'effettivo miglioramento di prestazioni poiché l'inizializzazione è necessaria in un utilizzo reale e se il miglioramento risultasse molto piccolo rispetto alla sola inizializzazione sarebbe poco significativo. I tempi sono misurati secondo il valore "real" del comando *time* disponibile nelle distribuzioni di GNU/Linux utilizzate e nello standard POSIX (e quindi in UNIX). I risultati nelle tabelle sono presentati utilizzando utilizzando la lettera "m" dopo i minuti, la lettera "s" dopo i secondi e il punto per separare i decimali secondo la notazione anglosassone, nei grafici invece si utilizza la virgola come separatore della parte frazionaria.

Per le tabelle che utilizzano abbreviazioni ci si riferisce a questa legenda:

R: GPU ATI Radeon HD 5830 (14 computation unit)

T: GPU Nvidia Tesla C1060 (30 computation unit)

G: GPU Nvidia GTS450 (4 computation unit)

C8: CPU Intel Core i7 930 (8 core logici su 4 core fisici)

C4: CPU Intel Core i7 930 (4 core fisici)

C2: CPU Intel Core i7 930 (2 core fisici)

C1S: CPU Intel Core i7 930 (1 core fisico + Simultaneous Multithreading = 2 core logici)

C1: CPU Intel Core i7 930 (1 core fisico)

DL: Dimensione locale (dimensione di uno work-group)

Si è iniziato dalla versione in doppia precisione poiché è spesso richiesta dalle applicazioni scientifiche, si sono effettuate in seguito prove con singola precisione per verificare il possibile aumento di prestazioni e le possibili divergenze.

5.2 Versioni in virgola mobile a doppia precisione

5.2.1 Versione C sequenziale con vettori di vettori per rappresentare matrici

Core i7 930	Phenom X4 9950	Xeon 5520
14m49.811s	20m0.775s	15m44.496s

Tabella 5.1: Tempi di esecuzione della versione sequenziale C con puntatori a puntatori in doppia precisione

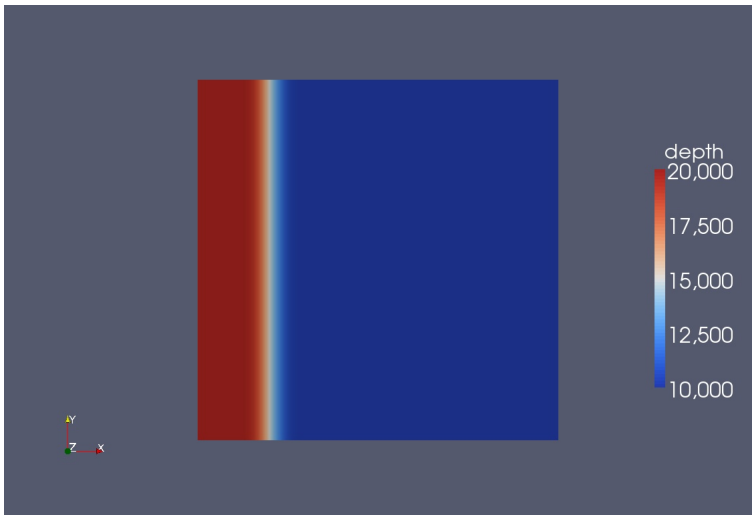


Figura 5.1: Esempio di output Paraview per una simulazione in doppia precisione di dimensione 1000x1000 e discretizzazione 0.5 al passo 1000 (t=5,050762722761≈5,051) di 3960. Nella legenda dell'immagine la virgola è usata come separatore della parte frazionaria.

Questa è la versione sequenziale utilizzata come riferimento per la correttezza delle successive implementazioni che sono tutte basate su questa. Come si può notare i processori utilizzati dispongono tutti di più core tuttavia il codice non è progettato per sfruttarli e utilizza quindi un solo core, il vantaggio della presenza di più core potrebbe essere notato nel caso si avviino contemporaneamente altre applicazioni di simulazione sequenziali computazionalmente pesanti e in questo caso la riduzione di prestazioni potrebbe essere particolarmente ridotta rispetto all'esecuzione multipla di tutte le istanze su singolo core (o singolo processore), tuttavia questo non è lo scopo di questa tesi. Il server Veio dispone di 4 processori Xeon quad-core tuttavia tale codice non è stato realizzato per sfruttare la presenza di più processori.

5.2.2 Versione C sequenziale con vettori per rappresentare matrici

Core i7 930	**Phenom X4 9950**	**Xeon 5520**
14m40.830s	19m42.350s	15m35.596s

Tabella 5.2: Tempi di esecuzione della versione sequenziale C con vettori puri in doppia precisione

Questa versione mostra un leggero miglioramento, anche se non molto rilevante, e sarà utilizzata come riferimento per i miglioramenti introdotti dalla parallelizzazione rispetto alla versione sequenziale infatti, come esposto in 4.1, in OpenCL è necessario utilizzare puntatori, che possono essere interpretati come *array* puri, ed è necessario decidere esplicitamente la posizione delle diverse celle dati i due indici affinché siano interpretate come celle di una matrice, e quindi il confronto sarà fatto con questa implementazione che utilizza la stessa struttura dei dati.

5.2.3 Prima versione C con OpenCL

Dimensione locale	**R**	**T**	**G**	**C8**
NULL (default)	1m39.719s	n.d.	n.d.	1m45.199s
Max	(16x16) 1m10.957s	(22x22) n.d.	(32x32)(1x512)(8x16) n.d.	(32x32) 1m20.697s
1x1	8m39.087s	n.d.	n.d.	15m25.625s
	C4	**C2**	**C1S**	**C1**
NULL (default)	1m57.731s	3m28.505s	5m0.425s	5m46.727
Max	n.d.	n.d.	n.d.	n.d.
1x1	n.d.	n.d.	n.d.	n.d.

Tabella 5.3: Tempi di esecuzione della prima parallelizzazione in doppia precisione utilizzante OpenCL

Si può notare un guadagno elevato: l'intero tempo di esecuzione su GPU Radeon è meno di 1/8 e poco più di 1/9 rispetto allla versione sequenziale su Core i7 e la versione OpenCL su Core i7 (che sfrutta gli 8 core) impiega meno di 1/8 del tempo rispetto alla versione sequenziale sullo stesso processore. In questa implementazione la GPU è leggermente meno performante della CPU. Non si è utilizzato il processore Xeon perché il server Veio non dispone di un runtime OpenCL che supporti l'esecuzione su CPU. Stessa cosa per la CPU Phenom e il computer Iperione. Per raggiungere il numero di 8 core sul modello di Core i7 utilizzato è necessario attivare l'Intel Simultaneous Multithreading, per valutare quanto la differenza tra 4 e 8 core sia dovuta all'elevato numero di Core da gestire e quanto dall'Intel SMT è stata effettuata anche una prova con 1 core attvo e SMT attivo. L'utilizzo della dimensione locale massima consentita (senza memoria locale) ha prodotto un leggero miglioramento di prestazioni mentre una dimensione ridotta a un solo work-item per ogni work-group ha moltiplcato significativamente il tempo di esecuzione. Per la Tesla i dati non sono disponibili a causa dell'uccisione del processo (NULL e 1x1) e del fallimento dell'allocazione di memoria (22x22). Per la GTS450 i tempi non sono disponibili a causa dell'uccisione del processo da parte del server o del fallimento dell'avvio (32x32).

5.2.4 Versione con copia parallela di righe e colonne

R	T	G	C8	C4	C2	C1S	C1
59.027s	2m38.069s	2m29.520s	2m4.900s	2m48.088s	5m25.386s	6m22.419s	10m20.454s

Tabella 5.4: Tempi di esecuzione della versione OpenCL in doppia precisione con copia parallela di righe e colonne

Il tempo risparmiato nell'esecuzione rispetto alla versione precedente con dimensione di default è del 40.81% nel caso di esecuzione su GPU Radeon (rispetto alla dimensione di default) mentre nel caso CPU le prestazioni sono peggiorate, aumentando il tempo di poco del 18.73% nel caso ad 8 core (rispetto alla dimensione di default). Per le GPU Nvidia non si può valutare il miglioramento poiché i tempi di esecuzione della versione precedente non sono disponibili. È interessante notare il miglioramento elevato di prestazioni della Tesla rispetto al processore Xeon (per il server Veio) e dal processore Phenom alla GTS450 per il computer Iperione.

5.2.5 Versione con uso di memoria locale e copie successive da memoria globale

Versione	R	T	G	C8
Controllo di flusso	1m7.765s	n.d.	n.d.	10m44.959s
Predicazione (operatore ternario)	1m8.743s	n.d.	n.d.	10m12.725s

Tabella 5.5: Tempi di esecuzione del primo tentativo di utilizzo di memoria locale in doppia precisione

Si può notare per la GPU Radeon un incremento prestazionale rispetto alla prima versione OpenCL pari al 32.04% anche se inferiore a quello ottenuto nella versione precedente (40.81%). Non si è ritenuto utile procedere con la verifica della scalabilità su un diverso numero di core per la CPU poiché il risultato è peggiore di quelli precedenti (il tempo è più di 5 volte quello della versione precedente, già alto in confronto alla prima versione). La versione utilizzante per il kernel la predicazione tramite l'operatore ternario ? : ha comportato tempo di esecuzione confrontabile sulla GPU e un leggero miglioramento sulla CPU, l'esecuzione su CPU rimane comunque particolarmente inefficiente. Non si è proceduto alla verifica di questa versione per la GTS450 e la Tesla C1060.

5.2.6 Versione con uso di memoria locale

R	T	G	C8
1m6.485s	n.d.	n.d.	9m43.698s

Tabella 5.6: Tempi di esecuzione del secondo modo di utilizzo di memoria locale in doppia precisione

Per la GPU Radeon si può notare un elevato guadagno prestazionale pari al 33.33% rispetto alla prima versione OpenCL, di poco superiore al primo modo di utilizzo della memoria locale e inferiore a quello ottenuto parallelizzando la copia di righe e colonne, quindi l'utilizzo della memoria locale migliora significativamente le prestazioni. Anche se la differenza di prestazioni con il caso precedente non fosse dovuta a una maggiore velocità del codice questa versione ha il vantaggio di avere un codice più chiaro e pulito che utilizza meno variabili private (quelle interne al kernel).

Come si può notare l'aggiunta dell'utilizzo di memoria locale in questo caso e nel caso precedente ha provocato un peggioramento alle performance dell'esecuzione su CPU, le cause possono essere:

- le copie verso la memoria locale costituiscono un'appesantimento inutile del codice del codice su CPU che a fronte di una copia in più non risparmia abbastanza tempo negli accessi successivi nel caso il tempo di accesso a memoria globale sia confrontabile con quello di accesso a memoria locale.
- il runtime di AMD (o le librerie per la compilazione) non sfrutta in maniera adeguata i processori Intel quando è introdotto l'uso di memoria locale

Si ritiene che la spiegazione principale sia la prima. Come è stato riportato anche l'implementazione di Intel non utilizza memoria locale. Non si è ritenuto utile procedere con la verifica della scalabilità su un diverso numero di core per la CPU a causa delle scarse prestazioni.

Il risultato per la GTS450 non è disponibile poiché il runtime OpenCL restituisce per ogni dimensione un massimo di 1024 e il programma fa la radice quadrata di entrambi scegliendo una dimensione di 32x32 con quindi la dimensione di uno work-group pari a 1024 ma al controllo del parametro

CL_KERNEL_WORK_GROUP_SIZE pari a 704 (il numero massimo di work-item in un work-group basato sulle risorse, come il numero di registri, richeste dal kernel) risulta superiore a quanto consentito. Per la Tesla è indicato allo stesso modo che la dimensione richiesta di 484 (il quadrato di 22, la parte intera della radice quadrata di 512 che è il massimo consentito per le prime due dimensioni) è superiore al massimo consentito che è 384. Si sono quindi utilizzate le versioni successive (eccetto la "versione con uso di memoria locale e copia parallela di righe e colonne") che hanno come alternative una dimensione ridotta a 1/4 del massimo o selezionabile con numeri a scelta (nella versione

con uso di memoria locale senza copia dei buffer con copia parallela di righe e colonne, considerando anche che è la più efficiente sulla GPU Radeon).

5.2.7 Versione con uso di memoria locale e copia parallela di righe e colonne

R	T	G	C8
27.695s	n.d.	n.d.	10m24.514s

Tabella 5.7: Tempi di esecuzione della versione in doppia precisione con utilizzo di memoria locale e copia parallela di righe e colonne

Come si può notare per la GPU Radeon la somma tra il vantaggio ottenuto rispetto alla prima versione OpenCL è meno che lineare tuttavia il vantaggio della versione con copia parallela di righe e colonne rispetto alla prima è di 110.994 secondi e quello della versione con memoria locale è di 103.536 secondi, la somma dei due valori è 214.53, abbastanza vicino al 142.326 (pari a quasi i 2/3 della precedente somma) di guadagno di questa versione (sempre rispetto alla prima versione OpenCL con dimensione di default). Per la GTS450 e la Tesla C1060 valgono le stesse considerazioni sulle dimensioni fatte nel caso precedente.

5.2.8 Versione con uso di memoria locale senza copia dei buffer

DL	R	T	G	C8
Max	(16x16) 55.540s	(22x22) n.d.	(32x32) n.d	(32x32) 9m19.660s
$\frac{1}{4}Max$	(8x8) 57.140s	(11x11) n.d.	(16x16) n.d.	(16x16) 3m30.450s

Tabella 5.8: Tempi di esecuzione della versione in doppia precisione con utilizzo di memoria locale senza copia dei buffer

Eliminando la distinzione tra lavori *old* e valori correnti le prestazioni sono leggermente migliori rispetto alla prima e alla seconda versione utilizzanti memoria locale ma inferiori rispetto alla versione che usa sia la memoria locale sia la copia di righe e colonne, questo accade perché il vantaggio derivante dall'eliminazione della copia dei dei valori dei buffer contenenti i valori correnti ai buffer old è piccolo rispetto alla necessità di risincronizzare i buffer per la copia sequenziale di righe e colonne. Riducendo a 1/4 del massimo la dimensione di uno work-group il tempo risultante sulla GPU è rimasto pressoché invariato, su

CPU invece il tempo si è ridotto significativamente rimanendo comunque peggiore della prima implementazione OpenCL (ma comunque pari a una frazione dell'esecuzione sequenziale). Per la GTS450 il caso 32x32 è fallito a causa di una dimensione superiore a 640, il caso 16x16 ha comportato l'uccisione del processo. Per la Tesla C1060 il caso 22x22 è fallito a causa di una dimensione superiore a 384, il caso 11x11 ha comportato l'uccisione del processo.

5.2.9 Versione con uso di memoria locale senza copia dei buffer con copia parallela di righe e colonne

DL	**R**	**T**	**G**	**C8**
Max	(16x16) 14.686s	(22x22) n.d.	(32x32) n.d	(32x32) 10m4.377s
$\frac{1}{4}Max$	(8x8) 14.613s	(11x11) 36.195s	(16x16) 35.988s	(16x16) 5m13.027s

Tabella 5.9: Tempi di esecuzione della versione in doppia precisione con utilizzo di memoria locale senza copia dei buffer e copia parallela di righe e colonne

Per la GPU Radeon la combinazione di tutti i metodi ha prodotto un tempo di esecuzione pari al 14.73% di quello della prima versione OpenCL (con dimensione di default) e all'1.67% di quello della versione sequenziale su Core i7, portando quindi a una versione più efficiente rispetto a tutte le precedenti. Riducendo la dimensione del work-group il tempo di esecuzione su GPU differisce di pochi centesimi di secondo, su CPU invece è ridotto a poco più della metà rimanendo comunque al di sopra della versione precedente. Per questa versione si sono valutate anche altre dimensioni degli work-group, in particolare le combinazioni non quadrate per generare uno work-group della dimensione massima di 256 elementi per la Radeon HD 5830 e le dimensioni 64x1 e 1x64 pari a un singolo wave-front

256x1	1x256	128x2	2x128	64x4	4x64	32x8	8x32	64x1	1x64
24.428s	18.514s	21.765s	15.814s	17.533s	15.514s	15.702s	14.844s	23.390s	15.394s

Tabella 5.10: Tempi di esecuzione in doppia precisione per Radeon HD 5830 per work-group (e memoria locale) non quadrati

Si può notare che tutte le esecuzioni sono peggiori del caso 16x16 (e 8x8) e solo le prestazioni della dimensione 8x32 sono paragonabili (leggermente peggio).

Per la GTS450 il caso 32x32 è fallito poiché è stata riportata una dimensione maggiore di 640. Il caso 16x16 ha invece funzionato correttamente e riportato un tempo migliore dell'unica altra versione funzionate su GTS450 (copia parallela

di righe e colonne). Qui di seguito sono state valutate per la GTS450 diverse dimensioni non quadrate di 640, 512 o 256 elementi (il caso 16x16 è già riportato) e 32 pari a un singolo *warp*. È inoltre riportato il caso 25x25 (625) che è la più grande dimensione quadrata inferiore a 640.

25x25	640x1	1x640
41.871s	1m39.218s	31.606s

512x1	1x512	256x2	2x256	128x4	4x128	64x8	8x64	32x16	16x32
1m25.097s	30.446s	1m2.600s	31.667s	47.298s	32.151s	43.029s	33.802s	42.188s	40.341s

256x1	1x256	128x2	2x128	64x4	4x64	32x8	8x32
1m12.293s	26.299s	52.224s	27.725s	40.671s	28.412s	38.209s	29.722s

32x1	1x32
1m6.876s	35.244s

Tabella 5.11: Tempi di esecuzione in doppia precisione per GTS450 per work-group (e memoria locale) non quadrati e 25x25

Si può notare che in generale le prestazioni sono migliori con una prima dimensione piccola e che l'esecuzione migliore è risultata quella con dimensione 1x256.

Per la Tesla C1060 il caso 22x22 è fallito poiché è stata riportata una dimensione maggiore di 384. Il caso 11x11 ha invece funzionato correttamente e riportato un tempo migliore dell'unica altra versione funzionate su Tesla (copia parallela di righe e colonne). Qui di seguito sono state valutate per la Tesla diverse dimensioni non quadrate di 512, 256 elementi e 32 pari a un singolo *warp*. È inoltre riportato il caso 19x19 (361) che è la più grande dimensione quadrata inferiore a 384. Infine è presente anche il caso 16x16 come dimensione quadrata per 256.

1x384	384x1	19x19
n.d.	n.d.	41.249s

256x1	1x256	128x2	2x128	64x4	4x64	32x8	8x32	16x16
n.d.	n.d.	59.414s	32.205s	50.189s	31.709s	52.128s	33.704s	40.885s

32x1	1x32
1m16.020s	33.935s

Tabella 5.12: Tempi di esecuzione in doppia precisione per Tesla C1060 per work-group non quadrati di dimensione 384, 256 e 32 e dimensioni quadrate 19x19 e 16x16

Le dimensioni non disponibili lo sono a causa del fallimento dell'allocazione di memoria. Come per i casi precedenti le prestazioni migliori si hanno con una prima dimensione piccola, la dimensione che ha comportato prestazioni migliori è stata invece la 4x64.

5.2.10 Versione con numero ridotto di work-item

DL	R	C8	C4	C2	C1S
NULL (default)	42m39.821s	1m43.059s	2m34.613s	3m19.569s	3m19.763s
1	37m43.673s	1m43.274s	n.d.	n.d.	n.d.
Max (numero di CU)	42m47.372s	3m19.343s	n.d.	n.d.	n.d.

Tabella 5.13: Tempi di esecuzione della versione utilizzante OpenCL con numero di work-item pari a quello di computation units (o core)

Le prestazioni indicando o no la dimensione della memoria locale sono pressoché equivalenti, solo l'utilizzo di un work-group di dimensione massima su CPU ha comportato un netto peggioramento, dal monitor di sistema e da questo è apparso chiaro che per la CPU l'utilizzo di una dimesione locale pari a quella globale ha forzato l'esecuzione principalmente su un solo core.

A causa delle scarse prestazioni sulla GPU Radeon non si sono effettuati test sulle altre GPU disponibili. L'elevata differenza di prestazioni su GPU rispetto alle altre versioni è da ricercare soprattutto nel ridotto numero di item che non sfrutta tutti gli stream core presenti in una compute unit della GPU, l'approccio dei due cicli for innestati è inoltre inefficiente per la GPU, è comunque l'unico approccio disponibile nel caso non si voglia associare uno work-item ad ogni cella.

I casi con 4 core o meno per la dimensione locale unitaria e massima non sono riportati perché nel primo caso sarebbero simili al caso di default e nel secondo sarebbero peggiori come appare chiaramente dal confronto con 8 core.

I risultati per 1 core non sono disponibili poiché per riuscire a coprire eventuali resti della divisione in sezioni si è scelto di dividere in sezioni di larghezza pari a $Nx/(numerodicore-1)$ che non è applicabile per un numero di core pari a 1. Il codice sarebbe comunque simile alla versione sequenziale e alla versione a due core che con questo approccio sfrutta solo uno dei due item avviati (poiché la divisione è per 1 l'altro item non effettua calcoli perché dovrebbe iniziare da oltre Nx e quindi non inizia il ciclo).

5.2.11 Scalabilità per dimensione della griglia a discretizzazione variabile

Per la Radeon HD si è utilizzata la versione con uso di memoria locale senza copia dei buffer con copia parallela di righe e colonne e dimensione 16x16, non si è uilizzato il caso 8x8 poiché a fronte di una riduzione delle dimensioni così elevata non si è ritenuto che un margine di pochi centesimi potesse essere conclusivo di un miglioramento rispetto alla scelta standard più semplice. Per la Radeon la dimensione 4000x4000 e le seguenti non sono disponibili a causa del fallimento dell'allocazione della memoria.

Per la Tesla C1060 si è utilizzata la dimensione 4x64 e il caso 5000x5000 non è disponibile per il fallimento dell'allocazione di memoria.

Per la GTS450 si è usata la dimensione locale 1x256, tranne nel caso 100x100 in cui si è usata la dimensione 4x64, la migliore con entrambi i lati inferiori a 100 (per completezza si riporta che con dimensione 1x256 si ottiene un tempo di esecuzione di 0.360s e con 16x16 di 0,348 s), il caso 5000x5000 non è disponibile a causa del fallimento dell'allocazione di memoria.

Per la CPU Core i7 si è utilizzata la prima versione OpenCL con dimensione massima degli work-group (32x32). Inoltre, anche se meno performante nel test di confronto, si è proceduto alla verifica della scalabilità della versione a ridotto numero di work-item con dimensione di default degli work-group (vers. 2) e unitaria (vers.3) ritenendo che potesse scalare meglio.

Gli altri risultati non disponibili sono dovuti alla lunghezza dell'esecuzione.

Il numero di passi non include il passo 0 che consiste nello riempimento predefinito della matrice Z e non è quindi uno dei passi di calcolo parallelo.

	dx	dt	N. passi	R	T	G	C8	C8 vers. 2	C8 vers. 3
100x100	5	0.050507627227611	396	0.790s	3.651s	0.347s	0.445s	0.460s	0.438s
500x500	1	0.010101525445522	1980	3.381s	7.485s	4.163s	9.566s	9.574s	9.664s
1000x1000	0.5	0.005050762722761	3960	14.686s	31.709s	26.299s	1m20.697s	1m43.059s	1m43.274s
2000x2000	0.25	0.002525381361381	7920	1m38.480s	3m41.613s	3m20.327s	11m54.912s	20m3.167s	20m5.164s
2500x2500	0.2	0.002020305089104	9900	3m11.861s	7m10.207s	6m29.113s	n.d.	n.d.	n.d.
4000x4000	0.125	0.001262690680690	15840	n.d.	28m50.867s	26m12.374s	n.d.	n.d.	n.d.
5000x5000	0.1	0.001010152544552	19799	n.d.	n.d.	n.d.	n.d.	n.d.	n.d.

Tabella 5.14: Prove di scalabilità a discretizzazione variabile in doppia precisione

	R	T	G	C8
100x100	0.00199494949495	0.00921969696970	0.00087626262626	0.00112373737374
500x500	0.00170757575758	0.00378030303030	0.00210252525253	0.00483131313131
1000x1000	0.00370858585859	0.00800732323232	0.00664116161616	0.00522651515152
2000x2000	0.01243434343434	0.02798143939394	0.02529381313131	0.09026666666667
2500x2500	0.01937989898990	0.04345525252525	0.03930434343434	n.d.
4000x4000	n.d.	0.10927190656566	0.0992660353534	n.d.
5000x5000	n.d.	n.d.	n.d.	n.d.

Tabella 5.15: Tempi di esecuzione del singolo passo per le prove di scalabilità a discretizzazione variabile in doppia precisione

Qui di seguito gli stessi risultati riportati in un grafico (sulle ascisse sono riportati i casi di test secondo il numero di celle di un lato della griglia):

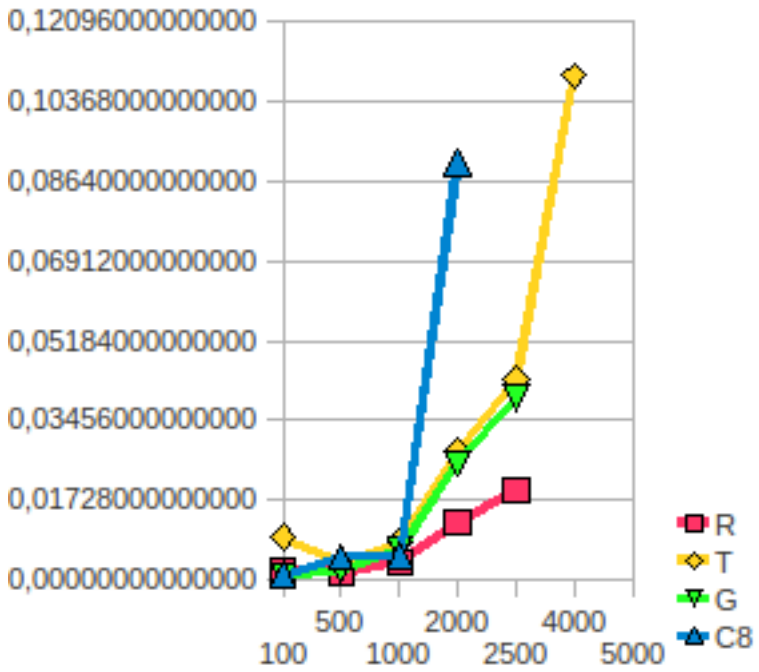


Figura 5.2: Grafico della scalabilità a discretizzazione variabile in doppia precisione

Il grafico indica chiaramente che la Radeon ha le prestazioni migliori anche se riesce a gestire una dimensione massima inferiore a quelle delle GPU Nvidia (che tra di loro sono circa equivalenti). La CPU ha invece avuto i tempi di esecuzione maggiori. L'unico comportamento non atteso sono state le prestazioni peggiori che la Tesla ha avuto con la dimensione 100x100 rispetto a 500x500.

5.3 Versioni a virgola mobile in singola precisione

5.3.1 Versione C sequenziale con vettori di vettori per rappresentare matrici

Core i7 930	Phenom X4 9950	Xeon 5520
18m48.564s	25m6.495s	20m25.477s

Tabella 5.16: Tempi di esecuzione della versione sequenziale C con puntatori a puntatori in singola precisione

5.3.2 Versione C sequenziale con vettori per rappresentare matrici

Core i7 930	Phenom X4 9950	Xeon 5520
18m35.766s	24m27.756s	20m4.427s

Tabella 5.17: Tempi di esecuzione della versione sequenziale C con vettori puri in singola precisione

5.3.3 Prima versione C con OpenCL

DL	R	T	G	C8
NULL (default)	1m7.377s	n.d.	n.d.	1m42.832s
Max	(16x16) 42.054s	(22x22) n.d.	(32x32)(1x512)(8x16) n.d.	(32x32) 1m22.445s
1x1	3m36.526s	n.d.	n.d.	15m19.133s
	C4	**C2**	**C1S**	**C1**
NULL (default)	2m13.184s	4m8.102s	5m47.149s	8m0.373s
Max	n.d.	n.d.	n.d.	n.d.
1x1	n.d.	n.d.	n.d.	n.d.

Tabella 5.18: Tempi di esecuzione della prima parallelizzazione in singola precisione utilizzante OpenCL

Si può notare come nella versione in doppia precisione un guadagno elevato rispetto all'esecuzione sequenziale (la versione per Radeon con dimensione di default impiega un tempo pari a meno di 1/16 rispetto alla versione sequenziale su Core i7) tuttavia in questa implementazione la GPU è leggermente meno performante della CPU. Non si è utilizzato il processore Xeon perché il server Veio non dispone di un runtime OpenCL che supporti l'esecuzione su CPU. Stessa cosa per la CPU Phenom e il computer Iperione. Rispetto alla versione in doppia precisione si può notare che, mentre le versioni sequenziali peggiorano passando alla singola precisione, nell'implementazione OpenCL le prestazioni migliorano nell'esecuzione su GPU, su CPU invece variano con il numero di core essendo comunque quasi invariate (rispetto alla versione a singola precisione) nel caso ad 8 core. L'utilizzo della dimensione locale massima consentita (senza memoria locale) ha prodotto un leggero miglioramento di prestazioni comunque percentualmente superiore a quello ottenuto nel caso in doppia precisione mentre una dimensione ridotta a un solo work-item per ogni work-group ha moltiplcato significativamente il tempo di esecuzione. Per la Tesla i dati non sono disponibili a causa dell'uccisione del processo (per tutte le dimensioni). Per la GTS450 i tempi non sono disponibili a causa dell'uccisione del processo da parte del server per tutte le dimensioni.

5.3.4 Versione con copia parallela di righe e colonne

R	T	G	C8	C4	C2	C1S	C1
46.138s	1m32.115s	2m40.704s	2m29.670s	3m55.913s	7m46.822s	9m6.790s	15m13.727s

Tabella 5.19: Tempi di esecuzione della versione OpenCL in singola precisione con copia parallela di righe e colonne

Nel caso della Radeon HD i risultati sono significativamente migliori rispetto alla prima versione OpenCL con dimensione di default come in doppia precisione (per la Radeon si ha un risparmio di tempo del 31.52%), e sempre come nel caso in singola precisione i risultati su CPU sono peggiorati rispetto alla versione precedente. È interessante notare il miglioramento elevato di prestazioni della Tesla rispetto al processore Xeon (per il server Veio) e dal processore Phenom alla GTS450 per il computer Iperione.

5.3.4.1 Versione con copia parallela e float4 di righe e copia parallela di colonne

Poiché la versione è in singola precisione è stato possibile utilizzare un approccio float4 per le righe.

R	**T**	**G**	**C8**	**C4**	**C2**	**C1S**	**C1**
45.745s	1m28.130s	2m36.113s	2m34.150s	4m2.093s	7m53.303s	6m29.934s	15m49.254s

Tabella 5.20: Tempi di esecuzione della versione OpenCL in singola precisione con copia parallela float4 di righe e copia parallela di colonne

I tempi sono del tutto confrontabili con la versione che non fa uso di *float4*, anche se questa versione dovrebbe sfruttare meglio la banda disponibile.

5.3.5 Versione con uso di memoria locale e copie successive da memoria globale

Versione	**R**	**T**	**G**	**C8**
Controllo di flusso	35.408s	n.d.	n.d.	9m7.865s
Predicazione (operatore ternario)	35.814s	n.d.	n.d.	9m8.110s

Tabella 5.21: Tempi di esecuzione del primo tentativo di utilizzo di memoria locale in singola precisione

Nel caso in doppia precisione per la GPU Radeon il miglioramento introdotto con la copia righe e colonne era superiore a quello introdotto con la memoria locale, in singola precisione accade il contrario, la causa possibile è la maggior dimensione dai dati in doppia precisione. La versione utilizzante per il kernel la predicazione tramite l'operatore ternario ? : ha comportato tempo di esecuzione confrontabile con la versione precedente sia su GPU sia su CPU. Non si è proceduto alla verifica di questa versione per la GTS450 e la Tesla C1060.

5.3.6 Versione con uso di memoria locale

R	T	G	C8
35.705s	n.d.	n.d.	7m49.585s

Tabella 5.22: Tempi di esecuzione del secondo modo di utilizzo di memoria locale in singola precisione

Come nel caso precedente questo modo di utilizzare la memoria locale diminuisce il tempo di esecuzione più della copia parallela di righe e colonne, infatti per la Radeon si ha una riduzione del 47.01%. Il tempo è leggermente superiore al caso precedente (in doppia precisione accadeva il contrario) ma come indicato anche nel caso in doppia precisione i valori sono così prossimi che non si può concludere una differenza di prestazioni e questa versione è stata preferita per essere unita agli altri metodi per la chiarezza del codice.

Il risultato per la GTS450 non è disponibile poiché il processo viene ucciso primo del completamento e lo stesso per la Tesla C1060.

5.3.7 Versione con uso di memoria locale e copia parallela di righe e colonne

R	T	G	C8
15.508s	27.311s	31.242s	9m15.897s

Tabella 5.23: Tempi di esecuzione della versione in singola precisione con utilizzo di memoria locale e copia parallela di righe e colonne

Il miglioramento su Radeon rispetto alla prima versione OpenCL è di 51.869 secondi, vicino alla somma di 21.239 e 31.672 (i miglioramenti dovuti rispettivamente all'introduzione della copia di righe e colonne e del secondo metodo di utilizzo di memoria locale) che è pari a 52.911.

5.3.8 Versione con uso di memoria locale senza copia dei buffer

DL	R	T	G	C8
Max	(16x16) 29.746s	(22x22) n.d.	(32x32) n.d.	(32x32) 8m49.259s
$\frac{1}{4}Max$	(8x8) 32.159s	(11x11) n.d.	(16x16) n.d.	(16x16) 3m51.434s

Tabella 5.24: Tempi di esecuzione della versione in singola precisione con utilizzo di memoria locale senza copia dei buffer

Come nel caso in doppia precisione l'eliminazione della gestione della differenza tra valori del passo precedente e valori correnti ha aumentato le prestazioni.

Riducendo a 1/4 del massimo la dimensione di uno work-group il tempo risultante sulla GPU è rimasto pressoché invariato, su CPU invece il tempo si è ridotto significativamente rimanendo comunque peggiore della prima implementazione OpenCL (ma comunque pari a una frazione dell'esecuzione sequenziale).

Per la GTS450 e la Tesla C1060 il processo è stato ucciso prima del completamento con entrambe le dimensioni.

5.3.9 Versione con uso di memoria locale senza copia dei buffer e copia parallela di righe e colonne

DL	**R**	**T**	**G**	**C8**
Max	(16x16) 8.657s	(22x22) 24.564s	(32x32) 27.153s	(32x32) 10m21.835s
$\frac{1}{4}Max$	(8x8) 11.255s	(11x11) 19.491s	(16x16) 16.766s	(16x16) 6m6.723s

Tabella 5.25: Tempi di esecuzione della versione in singola precisione con utilizzo di memoria locale senza copia dei buffer e copia parallela di righe e colonne

Nel caso Radeon il tempo di esecuzione è il 12.85% di quello della prima versione OpenCL (con dimensione di default) e lo 0.78% di quello della versione sequenziale su Core i7. Riducendo la dimensione del work-group il tempo di esecuzione su GPU differisce di pochi secondi secondo, su GPU invece è ridotto a poco più della metà rimanendo comunque al di sopra della versione precedente. Per questa versione si sono valutate, come per il caso in singola precisione, anche altre dimensioni degli work-group, in particolare le combinazioni non quadrate per generare uno work-group della dimensione massima di 256 elementi per la Radeon HD 5830 e le dimensioni 64x1 e 1x64 pari a un singolo wave-front

256x1	1x256	128x2	2x128	64x4	4x64	32x8	8x32	64x1	1x64
19.124s	7.655s	15.424s	8.130s	13.287s	8.169s	9.906s	8.315s	20.649s	8.992s

Tabella 5.26: Tempi di esecuzione in singola precisione per Radeon HD 5830 per work-group (e memoria locale) non quadrati

Si può notare che tutte le esecuzioni sono peggiori del caso 16x16 tranne 1x256 che è migliore di circa un secondo, 2x128, 4x64 e 8x32 (leggermente meglio) e 1x64 (leggermente peggio) che sono confrontabili. Questo risultato concorda con la guida di AMD che riporta un caso di utilizzo con due indici di un vettore *float* (interpretato come una matrice) e consiglia di utilizzare, per evitare conflitti, una dimensione unitaria per la dimensione corrispondente all'indice che viene moltiplicato per la larghezza della matrice (in questa tesi è la dimensione 0, cioè la prima). Si nota inoltre che i miglioramenti e peggioramenti non sono allineati a quelli della singola precisione infatti le dimensioni dei dati sono diverse.

A causa del miglioramento riscontrato dalla dimensione 32x32 alla dimensione 16x16 qui di seguito sono state valutate per la GTS450 diverse dimensioni non quadrate di 1024 elementi (il lato 32 è già nella tabella di confronto tra hardware diversi), 512, 256 sempre non quadrate (il caso 16x16 è già riportato) e 32 pari a un singolo *warp*.

1024x1	1x1024	512x2	2x512	256x4	4x256	128x8	8x128	64x16	16x64
1m24.904s	15.390s	53.503s	15.640s	38.465s	16.868s	30.991s	17.695s	28.025s	20.892s
512x1	1x512	256x2	2x256	128x4	4x128	64x8	8x64	32x16	16x32
1m15.108s	13.897s	45.381s	14.030s	30.781s	14.953s	25.735s	15.865s	23.716s	18.018s

256x1	1x256	128x2	2x128	64x4	4x64	32x8	8x32
1m17.370s	13.446s	39.091s	13.522s	26.696s	14.166s	22.928s	14.855s

32x1	1x32
1m6.372s	29.803s

Tabella 5.27: Tempi di esecuzione in singola precisione per GTS450 per workgroup (e memoria locale) non quadrati di dimensione 1024, 512, 256 e 32

Si può notare che in generale le prestazioni sono migliori con una prima dimensione piccola e che l'esecuzione migliore è risultata quella con dimensione 1x256 come nel caso della GPU di AMD anche se il dispositivo può gestire dimensioni superiori.

Qui di seguito sono state valutate per la Tesla diverse dimensioni non quadrate di 512, 256 elementi e 32 pari a un singolo *warp*. È inoltre riportato il caso 16x16 come dimensione quadrata per 256. Il caso 22x22 che è la più grande dimensione quadrata inferiore a 512 è già stato riportato nelle tabelle di confronto.

512x1	1x512	256x2	2x256	128x4	4x128	64x8	8x64	32x16	16x32
n.d.	n.d.	36.657s	11.538s	31.853s	12.271s	29.816s	14.081s	27.754s	21.901s

256x1	1x256	128x2	2x128	64x4	4x64	32x8	8x32	16x16
44.877s	11.519s	36.563s	10.670s	31.673s	11.647s	28.908s	13.870s	22.833s

32x1	1x32
46.964s	11.304s

Tabella 5.28: Tempi di esecuzione in singola precisione per Tesla C1060 per work-group non quadrati di dimensione 512, 256 e 32 e dimensione quadrata 16x16

Si può notare che in generale le prestazioni sono migliori con una prima dimensione piccola e che l'esecuzione migliore è risultata quella con dimensione 2x128 anche se il dispositivo può gestire dimensioni superiori.

5.3.10 Versione con numero ridotto di work-item

DL	**R**	**C8**	**C4**	**C2**	**C1S**
NULL (default)	23m4.645s	3m36.239s	4m51.532s	5m39.797s	5m42.598s
1	20m4.463s	3m37.460s	n.d.	n.d.	n.d.
Max (numero di CU)	23m4.590s	5m40.335s	n.d.	n.d.	n.d.

Tabella 5.29: Tempi di esecuzione della versione utilizzante OpenCL con numero di work-item pari a quello di computation units (o core)

Le prestazioni indicando o no la dimensione della memoria locale sono pressoché equivalenti, solo l'utilizzo di un work-group di dimensione massima su CPU ha comportato un netto peggioramento confermando i risultati esposti nel caso in singola precisione.

A causa delle scarse prestazioni sulla GPU Radeon non si sono effettuati test sulle altre GPU disponibili. L'elevata differenza di prestazioni su GPU rispetto alle altre versioni è da ricercare nel ridotto numero di item come già esposto nel caso in singola precisione. I casi con 4 core o meno per la dimensione locale unitaria e massima non sono riportati per gli stessi motivi riportati nel caso in doppia precisione.

I risultati per 1 core non sono disponibili per gli stessi motivi indicati nel caso in doppia precisione.

5.3.11 Scalabilità per dimensione della griglia a discretizzazione variabile

Per la Radeon HD si è utilizzata la versione con uso di memoria locale senza copia dei buffer con copia parallela di righe e colonne e dimensione 1x256 per

le dimensioni superiori a 256. Per la dimensione 100x100 è stata utilizzata inizialmente la dimensione 4x64 (la dimensione con prestazioni migliori che abbia entrambi i lati inferiori a 100), infatti nel caso una delle due dimensioni locali sia superiore al corrispondente Nx+2 o Ny+2 i programmi utilizzano come dimensione globale quella locale e non effettuano calcoli sulle celle non significative. Per completezza si è confrontata la dimensione 4x64 con la dimensione di 1x256 che ha comportato un tempo di 0.748 s e con 16x16 con cui si è ottenuto un tempo di 0.729 s.

Per la Tesla C1060 si è utilizzata la dimensione 2x128 (e 4x64 nel caso 100x100 per lo stesso motivo della GPU Radeon, per completezza si riporta che con 2x128 si ottiene un tempo di esecuzione pari 3.650 s, non molto diverso infatti la dimensione globale del lato rispettivo a 64 deve comunque essere portata a 128, e con 16x16 si hanno 3.461 s).

Per la GTS450 si è utilizzata la dimensione 1x256 (e 4x64 nel caso 100x100, per completezza si riporta che con dimensione 1x256 si ottiene un tempo di esecuzione di 0.356 s e con 16x16 si ottiene un tempo di 0.332 s).

Per la CPU Core i7 si è utilizzata la prima versione OpenCL con dimensione massima degli work-group (32x32). Inoltre, anche se meno performante nel test di confronto, si è proceduto alla verifica della scalabilità della versione a ridotto numero di work-item con dimensione di default degli work-group (vers. 2) e unitaria (vers.3) ritenendo che potesse scalare meglio.

I risultati non disponibili sono dovuti alla lunghezza dell'esecuzione.

Il numero di passi non include il passo 0 che consiste nello riempimento predefinito della matrice Z e non è quindi uno dei passi di calcolo parallelo.

	dx	dt	N. passi	R	T	G	C8	C8 vers. 2	C8 vers. 3
100x100	5	0.050507627427578	396	0.736s	3.826s	0.330s	0.433s	0.460s	0.445s
500x500	1	0.010101525112987	1980	2.660s	4.679s	2.426s	11.678s	34.061s	34.054s
1000x1000	0.5	0.005050762556493	3960	7.655s	10.670s	13.446s	1m22.445s	3m36.239s	3m37.460s
2000x2000	0.25	0.002525381278247	7920	39.175s	50.491s	1m43.253s	10m15.637s	23m45.214s	3m36.220s
2500x2500	0.2	0.002020305022597	9900	1m12.179s	1m33.088s	3m20.100s	n.d.	n.d.	n.d.
4000x4000	0.125	0.001262690639123	15840	4m30.494s	6m2.834s	13m37.331s	n.d.	n.d.	n.d.
5000x5000	0.1	0.001010152511299	19799	8m36.772s	11m52.423s	26m52.424s	n.d.	n.d.	n.d.

Tabella 5.30: Prove di scalabilità a discretizzazione variabile in singola precisione

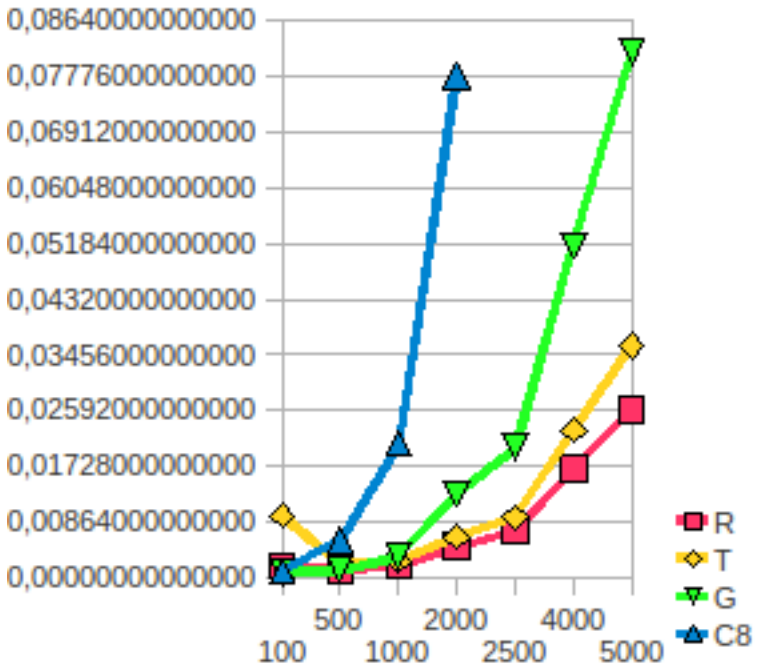


Figura 5.3: Grafico della scalabilità a discretizzazione variabile in singola precisione

	R	T	G	C8
100x100	0.00185858585859	0.00966161616162	0.00083333333333	0.00109343434343
500x500	0.00134343434343	0.00236313131313	0.00122525252525	0.00589797979798
1000x1000	0.00193308080808	0.00269444444444	0.00339545454545	0.02081944444444
2000x2000	0.00494633838384	0.00637512626263	0.01303699494950	0.07773194444444
2500x2500	0.00729080808081	0.00940282828283	0.02021212121212	n.d.
4000x4000	0.01707664141414	0.02290618686869	0.05159917929293	n.d.
5000x5000	0.02610091418759	0.03598277690792	0.08143966867013	n.d.

Tabella 5.31: Tempi di esecuzione del singolo passo per le prove di scalabilità a discretizzazione variabile in singola precisione

Innanzitutto si nota che le GPU riescono ora a gestire la dimensione 4000x4000 e anche le seguenti a causa della minor dimensione dei dati. Non sono disponibili dimensioni superiori a 5000x5000 perché già a partire da 6000x6000 il valore dt diventa troppo piccolo per essere gestito correttamente.

Qui di seguito gli stessi risultati riportati in un grafico (sulle ascisse sono riportati i casi di test secondo il numero di celle di un lato della griglia):

Le conclusioni non differiscono da quelle riportate per le prove in doppia precisione, compreso il miglioramento della Tesla da 100x100 a 500. L'unica differenza è stata una maggior distanza tra la Tesla e la GTS450 con prestazioni migliori della prima a partire dalla dimensione 2000x2000.

5.4 Riassunto

Sugli hardware utilizzati le implementazioni su GPU si sono rivelate più efficienti di quelle per la CPU tranne nell'implementazione OpenCL banale e nel caso di work-item ridotti.

Considerando per la dimensione 1000x1000 la migliore prestazione seriale su CPU (Core i7 con vettori) e la migliore esecuzione OpenCL (su Radeon HD con dimensione 8x8 per la doppia precisione e 1x256 per la singola precisione) si è ottenuto un miglioramento pari a 60.3 volte in doppia precisione e 145.8 volte in singola precisione.

La scalabilità ha evidenziato un normale decadimento delle prestazioni all'aumentare della dimensione tranne nel caso della Tesla che con una dimensione molto piccola (100x100) ha avuto un comportamento meno efficiente rispetto ad alcune dimensioni superiori.

Si nota infine che le GPU hanno avuto un comportamento decisamente peggiore nei calcoli in doppia precisione, la CPU invece ha mantenuto prestazioni simili per entrambe le precisioni.

Capitolo 6

Conclusioni

6.1 Considerazioni finali e confronto con altre implementazioni

Il lavoro ha evidenziato innanzitutto che OpenCL offre un modo relativamente semplice di programmare applicazioni parallele di tipo scientifico (in particolare quelle che operano su un dominio rappresentato come una griglia e utilizzano come dati di input solo valori di passi temporali precendenti) diminuendo di molto il tempo di esecuzione rispetto ad una implementazione sequenziale. Nel caso esposto le implementazioni banali che utilizzano OpenCL eseguite su diverse architetture (CPU e GPU) non offrono una significativa differenza di prestazione tra GPU e CPU, in particolare quelle di prezzo simile disponibili sul mercato nello stesso periodo (in questo caso Radeon HD 5830 e Core i7 930). Utilizzando poi accorgimenti specifici si è riuscito a migliorare ulteriormente le prestazioni solo su GPU.

OpenCL può essere quindi uno strumento potente e di facile utilizzo per parallelizzare applicazioni, tuttavia non è facile sfruttare al massimo le sue potenzialità, infatti nei casi esposti offre nelle implementazioni più banali vantaggi significativi ma confrontabili, sia su GPU sia su CPU.

Nel caso CPU c'è poco spazio per ulteriori miglioramenti invece nel caso GPU ci sono ulteriori margini di miglioramento rispetto a queste. Ciò è dovuto al fatto che la CPU ha dati locali e accesso alla RAM, la GPU invece dispone anche di una propria RAM e questo aggiunge un livello nella comunicazione con la RAM di sistema. Il modo principale per limitare la copia di dati dalla memoria di sistema a quella video nel caso di esecuzione su GPU è l'utilizzo della memoria locale, su CPU questo aumenta invece la complessità del codice degradando le prestazioni. Considerazioni sui conflitti possono poi far preferire una certa dimensione per gli work-group (in particolare la 1x256 con calcoli in singola precisione su Radeon HD della serie 5000 quando l'accesso ai dati è effettuato con `i*width+j`).

Le differenze tra CPU e GPU sono in linea con quelle di altri lavori come [Hagen et Al., 2005] (che però include anche la grafica) in cui sono utilizzati OpenGL e il linguaggio di Shading Cg ed è riportata una riduzione del tempo di esecuzione di 26.7 volte con il metodo Lax-Friedrichs e una dimensione di 1024x1024 (utilizzando schemi ad alta risoluzione tale lavoro ottiene risultati ancora migliori, vicini al metodo Lax-Friedrichs per dimensioni alte ma più elevati per dimensioni più basse). Considerando per la dimensione 1000x1000 la migliore prestazione seriale su CPU (Core i7 con vettori) e la migliore esecuzione OpenCL (su Radeon HD con dimensione 8x8 per la doppia precisione e 1x256 per la singola precisione) si è ottenuto in questa tesi un miglioramento pari a 60.3 volte in doppia precisione e 145.8 volte in singola precisione, ottenendo quindi un rapporto migliore, tuttavia per un confronto realmente significativo bisognerebbe considerare il costo dell'output grafico (il risultato è comunque incoraggiante).

Un'altro confronto interessante è con [Brodtkorb, 2010] (che utilizza CUDA C invece di OpenCL) che dichiara di poter gestire 530 milioni di celle in un secondo, quest'implementazione utilizza un'implementazione più evoluta rispetto al metodo Lax-Friedrichs inoltre gestisce anche il fondo asciutto e utilizza l'interoperabilità tra CUDA e OpenGL per la visualizzazione. L'implementazione riportata in questa tesi può gestire, in singola precisione con la Radeon HD 5830, 5000x5000 celle in 0.02610091418759 secondi cioè più di 957 milioni di celle in un secondo (va considerato però che i test non includono l'output). Considerando invece le dimensioni 500x500 e 1000x1000 (più vicine alle 480000 celle utilizzate nell'implementazione CUDA) si hanno rispettivamente più di 186 e più di 517 milioni di celle al secondo e quindi solo il secondo caso è paragonabile mentre il primo caso è meno performante (tuttavia i risultati sono incoraggianti infatti negli esempi qui riportati all'aumentare delle dimensioni si riesce a calcolare un numero superiore di celle per secondo e l'implementazione è migliorabile).

Altri lavori valutano le differenze tra CPU e GPU utilizzando implementazioni OpenCL per entrambi come in questa tesi, ad esempio lo fa [Blanchard] che include la grafica e misura le prestazioni in fps (*frames per second*).

6.2 Sviluppi futuri

Ulteriori sviluppi di questa tesi potrebbero riguardare diverse direzioni:

- per quanto riguarda la comprensibilità e la riusabilità del codice si potrebbe:
  - effettuare il refactoring del codice: nel caso il progetto debba essere esteso si potrebbero creare strutture dati che contengano il vettore (puntatore), il buffer OpenCL corrispondente (la variabile di tipo cl_mem) e le dimensioni della matrice in modo da ridurre i parametri delle funzioni. Questo semplificherebbe anche l'inclusione di un otuput grafico

  - produrre un porting a C++ utilizzando il binding conforme alle specifiche dello standard (OpenCL è uno standard per il linguaggio C e perciò è necessario un binding per C++) o quelli forniti da AMD o Nvidia (e anche Intel per le CPU), in questo caso si utilizzerebbero classi al posto delle strutture C per effettuare il refactoring del codice descritto al punto precedente. Il passaggio al C++ unito al refactoring faciliterebbe ulteriormente l'eventuale l'inclusione di un output grafico OpenGL o DirectX
  - rendere il codice portabile in maniera automatica su diverse piattaforme (quelle utilizzate in questa tesi e quelle descritte in seguito) e con scelte efficienti per ognuna (per questo dovranno essere sviluppati i punti seguenti riguardanti prestazioni e nuove piattorme)

- per aumentare le prestazioni si potrebbe:

  - valutare la possibilità di ulteriori miglioramenti alle prestazioni in particolare su CPU (Intel e AMD) e su GPU AMD e GPU Nvidia, utilizzando gli strumenti per la profilazione offerti da AMD, Nvidia e Intel o altri metodi
  - valutare se il caso in esame possa beneficiare di ulteriori approcci di tipo SIMD (le istruzioni SSE dei processori e anche la struttura delle GPU AMD dalla serie 5800 possono avvantaggiarsi di un approccio di questo tipo tuttavia il problema non sembra adatto a questa soluzione)

- per verificare le prestazioni di altre piattaforme hardware o software (l'utilizzo di piattaforme eterogenee o di altri modelli di parallelizzazione potrebbero poi anche aumentare le prestazioni) si potrebbe:

  - valutare le prestazioni per processori AMD, per le GPU AMD della serie FireStream per il calcolo ad alte prestazioni e le GPU NVIDIA per il mercato domestico della serie "Fermi" o successive
  - utilizzare più CPU o GPU (il runtime OpenCL, almeno nell'implementazione di AMD, gestisce in maniera trasparente più computation units e più core, anche logici, nel caso di CPU anche multiple ma non più GPU fisiche neanche se identiche, come riportato nell'AMD OpenCL University Kit) dove disponibile (producendo un programma che scali automaticamente su un numero maggiore di CPU o GPU o almeno scrivendo un codice che possa essere utilizzato per prove con un numero fissato di CPU o GPU su specifiche piattaforme). Tra più dispositivi nello stesso contesto (ad esempio più GPU dello stesso produttore) molti oggetti (kernel, programmi, oggetti di memoria) sono condivisi ma ogni dispositivo ha una sua *command queue* inoltre i trasferimenti di dati da un dispositivo all'altro devono passare obbligatoriamente dall'host (i dispositivi non posso effettuare direttamente i trasferimenti). Si possono usare contesti diversi e

librerie esclusive dell'host per la sincronizzazione e la comunicazione (ad esempio le primitive del sistema operativo): può essere una scelta alternativa a quella dell'unico contesto (ad esempio per riutilizzare conoscenze precedenti) con dispositivi identici dello stesso produttore ma è *necessario* in caso di sistemi eterogenei o distribuiti con nodi compatibili con OpenCL (usando ad esempio MPI nel caso distribuito)

- utilizzare CPU e GPU insieme, eventualmente sulle nuove architetture Fusion di AMD e "Sandy Bridge" di Intel dotate di sottosistema grafico che potrebbero permettere in futuro un approccio di questo tipo in modo più semplice, infatti queste architetture garantiscono la presenza di una CPU e una GPU dello stesso produttore, in particolare l'implementazione AMD di OpenCL supporta sia la parte CPU sia la parte GPU delle architetture Fusion. Attualmente AMD fornisce supporto per OpenCL sia a CPU (AMD e Intel) sia alle proprie GPU ma non fornisce librerie o funzioni specifiche per un approccio condiviso tuttavia è in teoria possibile realizzare programmi che sfruttino contemporaneamente CPU e GPU (anche se è necessario gestire esplicitamente i diversi dispositivi, bilanciare il carico di lavoro, sincronizzarlo e, come mostrato in questa tesi, utilizzare accorgimenti più specifici per le GPU). AMD è già al lavoro per creare un framework di questo tipo per le architetture Fusion, tali architetture potrebbero infatti avvantaggiarsi della condivisione della memoria evitando la replicazione dei dati, a questo proposito AMD fornisce a partire dall'AMD APP SDK 2.5 solo opzioni per la funzione *clCreateBuffer* per creare un buffer senza copie per il trasferimento di dati da CPU a GPU sfruttando il fatto che la memoria RAM sia condivisa tra GPU e CPU, l'utilizzo di questo metodo sfrutta la mancanza di limitazioni dovute al bus PCI Express [AMD, 2011b] (attualmente non è disponibile una banda elevata tra CPU e GPU interna al die e l'unico mezzo di comunicazione sono le copie in RAM, eliminate in questo approccio per migliorare ulteriormente le prestazioni, si nota poi che comunque la comunicazione da GPU a CPU è molto più lenta di quella inversa [Stokes, 2011], i processori Intel "Sandy Bridge" invece dispongono di un *ring bus* interno). Sempre a questo proposito ricercatori della North Carolina State University hanno prodotto un software per architetture con CPU e GPU su singolo chip (che condividano una cache L3 on-chip e memoria off-chip) che viene eseguito al momento dell'avvio di un'applicazione GPU generando un codice per CPU (basato sul kernel GPU da utilizzare) che è eseguito prima dell'esecuzione dei thread GPU, con l'effetto di sfruttare la CPU per recuperare maniera efficiente i dati necessari alla GPU nella cache condivisa L3 (sfruttando anche il pre-fetch a livello L2), unendo così la potenza elaborativa della GPU e la miglior latenza della CPU, ottenendo un miglioramento medio del 21.4% e picchi del 113%. La

versione più recente dell'Intel OpenCL SDK (1.1 beta) non supporta l'esecuzione su GPU, neanche quelle prodotte da Intel. Un'altro progetto che potrebbe favorire questa direzione è il progetto "Denver" di Nvidia che mira a integrare le proprie GPU con processori con una propria architettura ARM sul singolo chip (attualmente non è noto a quale segmento di mercato si rivolgeranno questi prodotti e se quindi coinvolgerà ad esempio la serie Tesla, eventualmente per eliminare la necessità di utilizzare CPU di altri produttori e aumentare le prestazioni rendendo la scheda più indipendente dal resto di sistema, o se si rivolgerà a nuovi prodotti specifici). La GPU ha un maggiore throughput e può nascondere la latenza, la CPU disponde invece di bassa latenza, cache e tempo di lancio minore, utilizzandole insieme si possono aumentare le prestazioni di picco rispetto al singolo dispositivo

- verificare le differenze di prestazioni tra GNU/Linux (con le implementazioni OpenCL dei produttori di hardware), Windows (con le implementazioni dei produttori di hardware) e Mac OS X (con l'implementazione Apple di OpenCL disponibile da Mac OS X 10.6 "Snow Leopard" ed eventualmente quelle dei produttori hardware nel caso forniscano implementazioni diverse da quella di Apple). Per uilizzare l'implementazione Apple è necessario utilizzare un percorso diverso per la libreria OpenCL utilizzando `#include <OpenCL/opencl.h>` invece di `#include <CL/cl.h>`, questo è già previsto nel codice host prodotto utilizzando la compilazione condizionale e verificando la piattaforma
- valutare eventualmente le differenze dell'esecuzione tra runtime a 32 bit (su sistemi operativi a 32 bit ed eventualmente a 64 bit) e runtime a 64 bit su sistemi a 64 bit (in particolare per l'esecuzione su CPU in doppia precisione)
- utilizzare altri runtime OpenCL ed SDK per confrontare le prestazioni su CPU, ad esempio l'Intel OpenCL SDK (che è in grado di eseguire kernel solo su CPU e utilizza automaticamente un approccio SIMD quando possibile e può inoltre sfruttare al meglio il Simultaneous Multithreading di Intel)
- utilizzare implementazioni non OpenCL per il GPU computing facendo il porting a C for CUDA (supportato dalle GPU Nvidia) e a DirectCompute (parte delle DirectX 11 e quindi disponibile sulle GPU che le supportano) e fare il confronto di questi (sia a livello di semplicità sia a livello di efficienza) con OpenCL sulla stessa GPU
- utilizzare implementazioni non OpenCL che sfruttino CPU parallele, come OpenMP, per confrontare il comportamento della stessa CPU rispetto a OpenCL
- utilizzare il nuovo hardware Knights Ferry di Intel (se disponibile e supportato dall'implementazione OpenCL di Intel)

- per aumentare il realismo, l'utilità e l'utilizzabilità della simulazione si potrebbe:
  - inserire un output grafico con OpenGL (per GNU/Linux, in particolare per i server del Dipartimento di Matematica, o per tutte le piattaforme supportate) e/o Direct3D (parte di DirectX, solo per Windows) oppure utilizzando librerie di più alto livello come VTK (la stessa utilizzata dal software Paraview)
  - utilizzare le tecniche impiegate in questa tesi (e in quelle che eventualmente si baseranno su questa) per migliorare le prestazioni di un'applicazione esistente utilizzabile per altre simulazioni oltre a quella delle acque basse
  - creare una sola versione che includa tutte le scelte utilizzate scegliendo la versione da impiegare e tutti gli altri parametri in modo interattivo, con input da file o parametri da linea di comando
  - eliminare i limiti del codice su gestione delle zone asciutte e fondo piatto
  - valutare casi più realistici (eventualmente dopo alcuni dei miglioramenti riportati in precedenza e sicuramente dopo aver inserito la gestione del fondo non piatto e delle zone asciutte)

Bibliografia

[Agoshkov, 1994] V.I. Agoshkov, E. Ovchinnikiv, A. Quarteroni, F. Saleri, "Recent developments in the numerical simulation of shallow water equations", Mathematical Models and Method in Appied Science, 4:533

[AMD, 2010a] Advanced Micro Devices, An Introduction to OpenCL™ http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/opencl-intro.aspx

[AMD, 2010b] Advanced Micro Devices, A Brief History of General Purpose (GPGPU) Computing http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/gpgpu-history.aspx

[AMD, 2010c] Advanced Micro Devices, ATI Stream SDK OpenCL™ Programming Guide (v1.05) http://developer.amd.com/gpu/ATIStreamSDK/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdfhttp://developer.amd.com/gpu/amdappsdk/assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf

[AMD, 2011a] Advanced Micro Devices, AMD Accelerated Parallel Processing OpenCL™ Programming Guide (v1.2) http://developer.amd.com/gpu/ATIStreamSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

[AMD, 2011b] Advanced Micro Devices, CPU-to-GPU data transfers exceed 15GB/s using APU zero copy path
http://blogs.amd.com/developer/2011/08/01/cpu-to-gpu-data-transfers-exceed-15gbs-using-apu-zero-copy-path

[BDT, 2011] Brown Deer Technology, "BDT Nbody Tutorial" http://www.browndeertechnology.com/docs/BDT_OpenCL_Tutorial_NBody-rev3.html

[Blanchard] Eric Blanchard, "OpenCL Implementation of Fluid Simulation" http://www-etud.iro.umontreal.ca/~blancher/projects/cl_fluids/rapport.pdf

[Brodtkorb, 2010] André R. Brodtkorba, Martin L. Sætrab, Mustafa Altinakarc, "Efficient Shallow Water Simulations on GPUs: Implementation, Visualization, Verification, and Validation" http://www.serri.org/publications/Documents/Ole%20Miss%20Project%2080031%20-%20Efficient%20Shallow%20Water%20Simulations%20on%20GPUs%20-%20August%202010%20(Brodtkorb,%20Saetra,%20Altinakar).pdf

[Charlier, 1982] R.H. Charlier, "Tidal Energy", Van Nostrand Reinhold Company, New York, NY

[Count, 1980] B. Count (editor), "Power from Sea Waters", Academic Press, London

[Dawson and Mirabito, 2008] Clint Dawson and Christopher M. Mirabito, "The Shallow Water Equations", Institute for Computational Engineering and Sciences University of Texas at Austin users.ices.utexas.edu/~arbogast/cam397/dawson_v2.pdf

[Fontana, 1998] Luca Fontana, "Un modello quasi idrostatico per la simulazione di correnti idrodinamiche", Tesi di laurea, Politecnico di MIlano, 1998

[Gray and Gashaus, 1972] T.J. Gray and O.K. Gashaus (editors), "Tidal Power", Plenum Press, New York, NY

[Hagen et Al., 2005] T.R. Hagen, J.M. Hjelmervik, K.–A. Lie, J.R. Natvig, M. Ofstad Henriksen, "Visual Simulation of Shallow-Water Waves", Simulation Modelling Practice and Theory 13, 716-725 http://www.math.sintef.no/gpu/pdf/visualwave.pdf

[Kawahara and Yoshida, 1978] M. Takeuchi Kawahara, N. and T. Yoshida, "Two Step Explicit Finite Element Method for Tsunami Wave Propagatin Analysis", Int. J. Numer. mEth. Eng., 2, 331-351

[Kinnmark, 1996] I. Kinnmark, "The Shallow Water Wave Equations: Formulation, Analyisys and Application", Lectures Notes in Engineering 15, Springer-Verlag Berlin Heidelberg New York Tokyo

[Kubatko, 2005] E. J. Kubatko: Development, Implementation, and Verification of hp-Discontinuous Galerkin Models for Shallow Water Hydrodynamics and Transport, Ph.D. Dissertation (2005)

[Kirk and Hwu, 2010] David B. Kirk, Wen-mei W.Hwu, "Programming Massively Parallel Processors - A Hands-on Approach", Elsevier Morgan Kauffmann Publishers

[Leendertse, 1967] J.J. Leendertse "Aspects of a Computational Model for Long Period Water-Wave Prpagation", Rand Memorandum RM-5249-PR, Santa Monica, CA

[Leveque, 2002] Randall J. Leveque, "Finite Methods for Hyperbolic Problems", Cambridge University Press

[NCSU, 2012] North Carolina State University, "Engineers Boost Computer Processor Performance By Over 20 Percent" `http://news.ncsu.edu/releases/wmszhougpucpu/`

[Oden, 2006] J. T. Oden: A Short Course on Nonlinear Continuum Mechanics, Course Notes

[Panton, 2006] R. L. Panton: Incompressible Flow, Hoboken, NJ: Wiley

[Politecnico di Milano, 2011] Politecnico di Milano - MOX Laboratorio di Calcolo, "VEIO: un server GPU" `http://mox.polimi.it/it/progetti/unix/gpu.php?en=`

[Sanders and Kandrot, 2010] Jason Sanders, Edward Kandrot, "CUDA by example - An introudction to General-Purpose GPU Programming", Addison-Wesley Pearson Education

[Stokes, 2011] Jon Stokes, "Why memory is the weak link in AMD's latest Fusion chip" `http://arstechnica.com/business/news/2011/06/another-look-at-amds-llano.ars`

[Toro, 2001] Eleuterio F. Toro, "Shock-Capturing Methods for Free-Surface Shallow Flows", John Wiley & Sons. Ltd.

[Wood, 1993] W.L. Wood, "Introduction to Numerical Methods for Water Resources", Cleradon Press, Oxford