UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VINÍCIUS GARCIA PINTO

Ambientes de Programação Paralela Híbrida

Trabalho Individual I
TI-XX

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, dezembro de 2011

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ALU	Arithmetic Logic Unit
API	Application Programming Interface
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
EIB	Element Interconnect Bus
FPGA	Field Programmable Gate Arrays
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HMPP	Hybrid Multicore Parallel Programming
KAAPI	Kernel for Adaptative Asynchronous Parallel and Interactive programming
MFC	Memory Flow Controller
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PPE	PowerPC Processing Element
SIMD	Single Instruction Multiple Data
SPE	Synergistic Processing Element
SPU	Synergistic Processing Unit
STL	Standart Template Library
SIMD	Single Instruction Multiple Data
TBB	Threading Building Blocks

LISTA DE FIGURAS

LISTA DE CÓDIGOS

RESUMO

Os limites atingidos pelas arquiteturas de computadores tradicionais baseadas em processadores sequenciais, têm incentivado a adoção de processadores *multicore* e aceleradores como forma de aumentar o desempenho dos sistemas computacionais atuais. Atualmente, os sistemas paralelos de alto desempenho fazem uso simultâneo de recursos de processamento distintos como CPUs *multicore* e GPUs. O uso de recursos de processamento distintos torna complexa a programação eficiente de sistemas híbridos. Dessa forma torna-se necessário o uso de ferramentas que auxiliem a programação, simplificando e tornando mais eficiente o uso dos recursos do sistema. Neste trabalho é apresentada uma contextualização sobre a programação paralela em arquiteturas híbridas. É apresentada uma visão geral sobre as arquiteturas paralelas híbridas atuais e são apresentadas duas classes de ferramentas de programação que visam simplificar a programação e a execução de aplicações paralelas nessas arquiteturas.

Palavras-chave: Programação paralela, programação paralela híbrida, GPU, ferramentas para programação paralela.

ABSTRACT

The limits achieved by traditional computer architectures based on sequencial processors, have encouraged the use of multicore processors and accelerators as a way to increase the performance of computer systems. Today, high performance parallel systems use distinct processing resources, such as multicore CPUs and GPUs. The use of distinct processing resources becomes efficient programming of hybrid systems complex. Therefore it becomes necessary the use of tools to simplify the programming and make more efficient use of system resources. In this work, we present a background about parallel programming on hybrid architectures. We present a overview about hybrid parallel architectures and two classes of programming tools that aim to simplify programming and execution of parallel applications in this architectures.

Keywords: parallel programming, hybrid parallel programming, GPU, parallel programming tools.

1 INTRODUÇÃO

Durante vários anos os sistemas paralelos estiveram limitados a arquiteturas multiprocessador, máquinas vetoriais e aglomerados, ou seja, equipamentos especializados e geralmente de acesso restrito às instituições de pesquisa e grandes empresas. Nesse período a evolução do desempenho nos computadores de uso geral baseou-se na diminuição da densidade dos transistores e no aumento da frequência de *clock* do processador (sequencial) (SHAN, 2006). Porém esse modelo de aumento de desempenho, baseado no processador sequencial, atingiu limites físicos que inviabilizaram sua continuidade e a solução adotada desde então tem sido o uso de arquiteturas com múltiplos núcleos de processamento (*multicore*) (ASANOVIC et al., 2009). Com isso, no cenário atual, tem-se a popularização de recursos de computação paralela, que tiveram seu custo reduzido e já estão presentes desde as mais simples estações de trabalho (RAUBER; RÜNGER, 2010).

Quase que simultaneamente à popularização dos processadores *multicore* ampliou-se o desenvolvimento de coprocessadores especializados, com desempenho otimizado para determinado tipo de operação. Tais processadores são empregados juntamente a processadores de uso geral, de forma a atuarem como aceleradores para alguns tipos restritos de operações. As unidades de processamento gráfico (GPU - *Graphics Processing Unit*) são exemplos de coprocessadores especializados utilizados como aceleradores no cálculo de operações gráficas. Embora projetadas inicialmente para cálculos gráficos, as GPUs atuais podem ser vistas como um tipo de *stream processor*[1] sendo utilizadas para alguns cálculos de propósito geral como operações sobre matrizes.

Atualmente os sistemas computacionais para processamento de alto desempenho têm sido construídos de forma híbrida, fazendo uso tanto de processadores *multicore* quanto de unidades de processamento gráfico (GPUs). A programação nessas arquiteturas individualmente tem sido feita através de APIs (*Application Programming Interface*), bibliotecas e ambientes de programação específicos para cada arquitetura. Nas arquiteturas *multicore* pode-se citar como exemplo as ferramentas Cilk (BLUMOFE, 1996), OpenMP (*Open Multi-Processing*) (CHAPMAN; JOST; PAS, 2007) e a biblioteca Intel TBB (*Threading Building Blocks*) (REINDERS, 2007). Quando refere-se a programação de GPUs são utilizados ambientes de programação como CUDA (Compute Unified Device Architecture) (SANDERS; KANDROT, 2010) e OpenCL (GASTER et al., 2011). No entanto, esses ambientes são voltados a apenas umas das arquiteturas *multicore* ou GPUs. Além disso, no caso da arquiteturas GPU a programação é feita em nível bastante próximo ao *hardware* o que torna o código restrito a determinado modelo/fabricante de *hardware* e por consequência pouco portável.

[1] Em um *stream processor*, dado um conjunto de dados (*stream*), uma série de operações (*kernel*) são aplicadas a cada um dos elementos do *stream*.

Ferramentas como ambientes de programação que ofereçam suporte simultaneamente à arquiteturas *multicore* e GPU podem extrair proveito tanto do processador de uso geral quanto do coprocessador especializado, utilizando-os de forma cooperativa. Nesse cenário, as tarefas de um fluxo de trabalho possam ser alocadas entre as unidades de processamento de diferentes tipos de forma que cada uma das unidades execute aquelas tarefas para as quais são mais adequadas (SHAN, 2006; HERMANN et al., 2010). Considera-se importante também que essas soluções simplifiquem o uso dos aceleradores para a programação de propósito geral e que mantenham o código portável entre diferentes modelos e/ou fabricantes do *hardware* (DOLBEAU; BIHAN; BODIN, 2007). Pode-se citar como exemplos de ambientes de programação paralela que suportam arquiteturas híbridas (CPUs *multicore* e GPUs) as ferramentas XKaapi e StarPU.

Esse trabalho apresenta uma contextualização da programação e gerenciamento da execução de aplicações paralelas em sistemas paralelos compostos de recursos heterogêneos. As demais seções deste trabalho estão organizadas da seguinte forma: o Capítulo 2 apresenta uma visão geral sobre os recursos de processamento utilizados nas arquiteturas paralelas híbridas nas quais se baseiam sistemas paralelos atuais. O Capítulo 3 apresenta ferramentas utilizadas para a programação dos aceleradores e coprocessadores que compõe as arquiteturas híbridas atuais. O Capítulo 4 apresenta ambientes de execução para aplicações paralelas que suportam arquiteturas híbridas. Por fim, o Capítulo 5 apresenta as considerações finais deste trabalho.

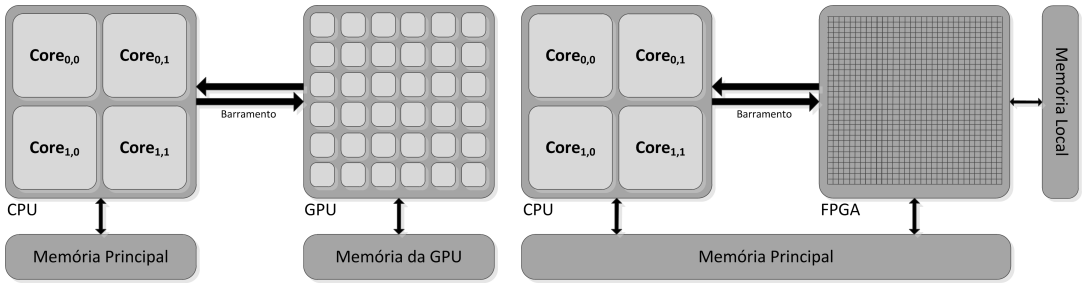
2 ARQUITETURAS PARALELAS HÍBRIDAS

Os limites atingidos pelas arquiteturas tradicionais juntamente com o advento de co-processadores especializados, também conhecidos como aceleradores e com a popularização dos processadores *multicore* têm incentivado o uso de sistemas computacionais paralelos com arquiteturas híbridas. Esses sistemas aliam unidades de processamento de propósito geral, por exemplo processadores *multicore*, à recursos de *hardware* especializados em certos tipos de operações,como FPGAs e GPUs (GASTER et al., 2011; LASTOVETSKY; LASTOVETSKY; DONGARRA, 2009). A Figura 2.1 apresenta dois exemplo de arquituras paralelas híbridas compostas de unidades de processamento de propósito geral (CPUs *multicore*) e de aceleradores (GPUs e FPGAs).

2.1 Processadores *multicore*

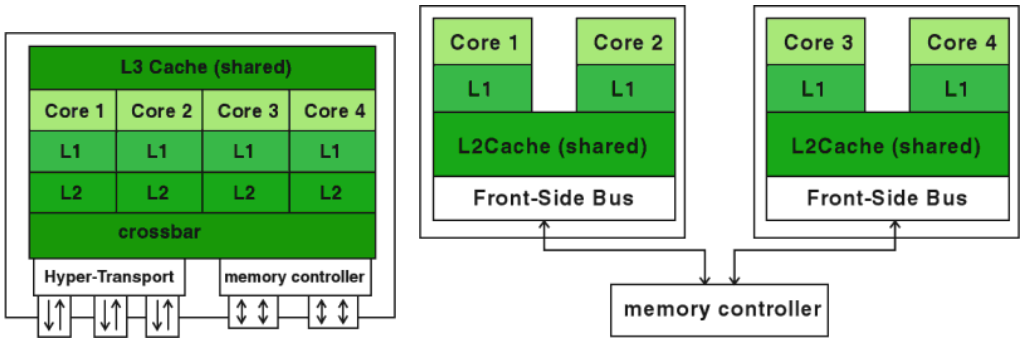
Os processadores *multicore* ou de múltiplos núcleos são unidades de processamento de propósito geral que integram dois ou mais núcleos completos de processamento em um mesmo *chip*. Essa arquitetura de processadores tornou-se popular nos últimos anos devido ao esgotamento do modelo de aumento de desempenho do processador sequencial (*singlecore*) que foi baseado no aumento da frequência de *clock*.

Do ponto de vista do sistema operacional, cada núcleo de um processador *multicore* é interpretado como um processador lógico independente com recursos de execução distintos, de forma que cada núcleo pode executar aplicações distintas. Além disso, através do uso de técnicas de programação paralela é possível executar uma aplicação em um conjunto de núcleos paralelamente buscando reduzir o tempo de execução em relação a um processador sequencial.



(a) CPU-GPU (b) CPU-FPGA

Figura 2.1: Exemplo de arquiteturas híbridas que combinam CPUs *multicore* com co-processadores especializados



(a) AMD Opteron (b) Intel Quad-Core Xeon

Figura 2.2: Arquitetura interna dos processadores *multicore* AMD Opteron e Intel Quad-Core Xeon (RAUBER; RÜNGER, 2010)

A arquitetura interna de um processador *multicore* pode ser organizada de forma hierárquica onde os múltiplos núcleos se comunicam por meio do compartilhamento de memórias *cache*. O nível de memória *cache* onde ocorre o compartilhamento de dados entre os núcleos varia conforme o fabricante e a tecnologia utilizada no processador. Na Figura 2.4 podem ser observadas duas abordagens distintas, no processador apresentado em 2.2(a) o compartilhamento ocorre na *cache* de nível 3, os núcleos possuem *caches* L1 e L2 privadas, uma outra abordagem é apresentada em 2.2(b) onde os núcleos compartilham dois a dois a memória *cache* L2 (RAUBER; RÜNGER, 2010).

A programação em processadores *multicore* tem sido feito por meio de ferramentas como:

- OpenMP (CHAPMAN; JOST; PAS, 2007) baseado na adição de diretivas de compilação em códigos C, C++ e Fortran para indicar a paralelização de laços ou definir algumas seções de código como *tasks* passíveis de serem executadas em paralelo;
- Intel TBB (REINDERS, 2007): é uma biblioteca de templates para programação paralela utilizando tarefas em C++. O escalonamento entre as tarefas paralelas é baseado na política *work stealing*;
- Cilk (BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998): é uma linguagem para programação paralela baseada em uma extensão da linguagem C que utiliza a política de escalonamento *work stealing*. A principal diferença no código Cilk em relação a código C são as palavras chave *spawn* (indica que a chamada procedimento será executada em paralelo) e *sync* (indica uma barreira para esperar que todos os procedimentos em paralelo tenham terminado sua execução).

2.2 Aceleradores

2.2.1 FPGA

Os *Field Programmable Gate Arrays* (FPGAs) consistem em dispositivos de *hardware* reconfiguráveis que podem ser reprojetados pelo programador para resolver de maneira mais eficiente alguns tipos específicos de problemas (SHAN, 2006).

Esses dispositivos têm sido utilizados a vários anos em aplicações como lógica discreta e processamento de sinais, recentemente também passaram a ser utilizados como aceleradores para aplicações de alto desempenho.

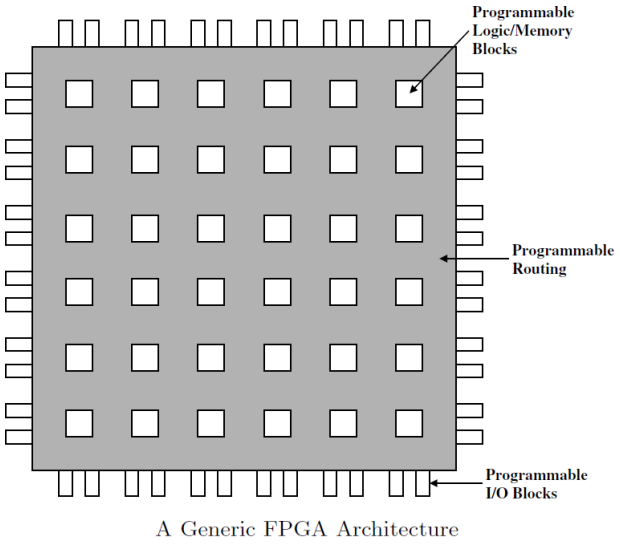


Figura 2.3: Arquitetura simplificada de um dispositivo FPGA

Um FPGA típico consiste em um conjunto de três tipos de elementos: blocos lógicos programáveis e de memória, roteamento e blocos de entrada e saída. A Figura 2.3 apresenta um exemplo simplificado de uma arquitetura FPGA. Uma arquitetura FPGA, em geral, é composta por um arranjo de blocos programáveis conectados por uma rede de interconexão flexível e reconfigurável (GOKHALE; GRAHAM, 2005).

A configuração e a programação de um dispositivo FPGA difere da programação de outros recursos de processamento como processadores *multicore* ou GPUs. Os FPGAs tradicionalmente são programados utilizando linguagens de descrição de *hardware* ou HDLs (*Hardware Description Languages*) que possibilitam a descrição do comportamento dos blocos e circuitos lógicos. Linguagens como VHDL e Verilog são exemplos de HDLs largamente utilizadas para configuração e programação do *hardware* de FPGAs. A programação com HDLs demanda por conhecimento e treinamento em projeto de *hardware*. Entretanto uma vez que uma aplicação específica tenha sido descrita e otimizada seu desempenho em um FPGA é bastante satisfatório.

2.2.2 GPU

As unidades de processamento gráfico (GPUs) foram inicialmente desenvolvidas e utilizadas para acelerar cálculos para computação gráfica 3D. Cálculos gráficos possuem características como: necessidade de grande poder computacional, permitem alto paralelismo e demandam alta largura de banda, ainda que com alta latência. Entretanto, essas características computacionais também estão presentes em muitas outras aplicações, o que incentivou a evolução e o uso de GPUs para outros cálculos de proposito geral e não apenas cálculos gráficos (OWENS et al., 2008).

GPUs consistem de um grande número de pequenos núcleos de processamento que trabalham de forma massivamente paralela permitindo que o desempenho de algumas aplicações em GPUs seja significativamente superior do que em uma CPU tradicional (NVIDIA, 2011a). Uma GPU é um *symmetric multicore processor* que é exclusivamente acessado e controlado por uma CPU. A GPU opera assincronamente em relação à CPU possibilitando execução e transferência de memória de forma concorrente.

As GPUs NVidia Fermi são um exemplo de arquitetura GPU com suporte à execução de cálculos de propósito geral. Nessa arquitetura estão disponíveis 512 núcleos de

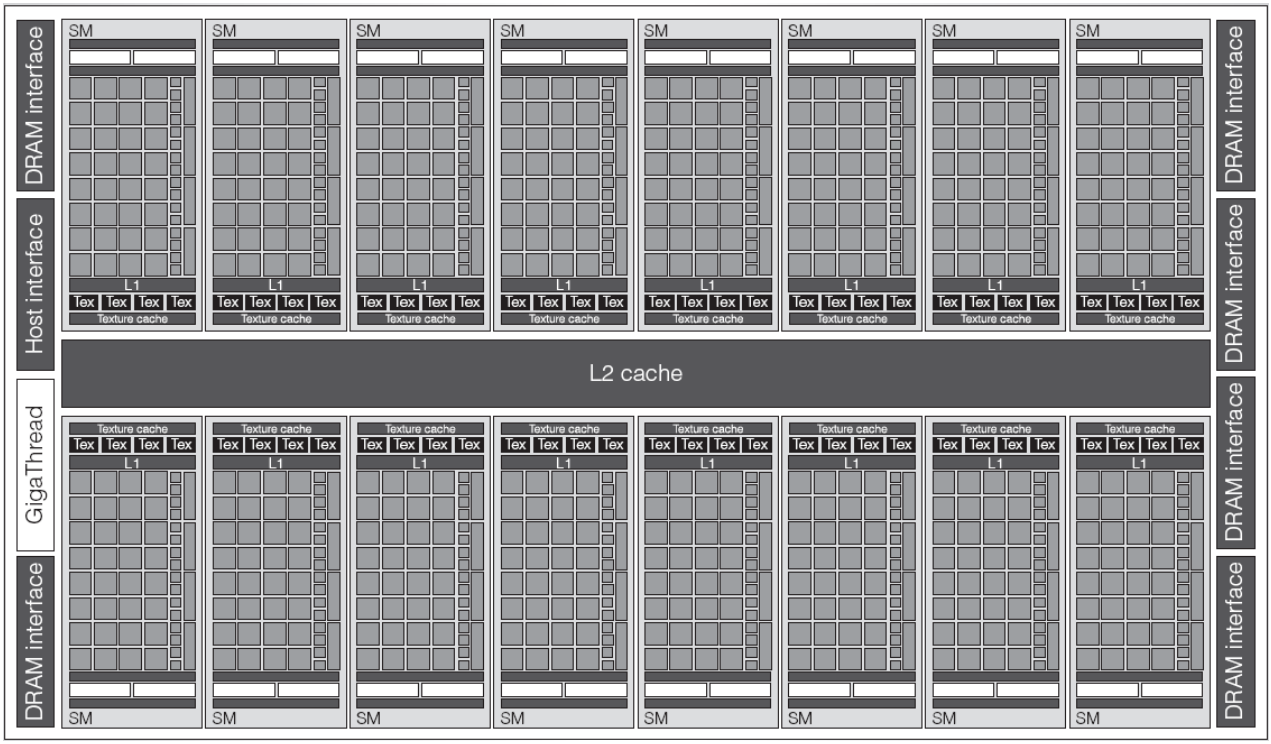


Figura 2.4: Arquitetura da GPU Nvidia Fermi

processamento encapsulados em 16 *streaming multiprocessors* (SMs). Cada *streaming multiprocessor* possui 32 núcleos, chamados *streaming processors*, e uma memória *cache* L1 privada, os 16 *streaming multiprocessors* compartilham uma *cache* L2 e o barramento de conexão com a CPU. Cada um desses núcleos de processamento pode executar uma instrução de ponto flutuante ou de ponto fixo por ciclo (NVIDIA, 2009; NICKOLLS; DALLY, 2010). A Figura 2.4 apresenta a organização interna de uma GPU NVidia Fermi.

A programação de propósito geral em GPUs tem sido feita através de modelos como CUDA e OpenCL. Esses modelos oferecem ao programador uma abstração pequena do *hardware* da GPU, logo tarefas como as alocações e transferências de memória devem ser indicadas explicitamente pelo programador. No Capítulo 3 são apresentadas ferramentas para programação em GPU, como CUDA, OpenCL, Thrust e ferramentas baseadas em diretivas de compilação.

2.3 Processadores Heterogêneos

2.3.1 Cell BE

A arquitetura de processadores Cell BE (Cell Broadband Engine) é uma arquitetura de processadores *multicore* heterogênea composta por nove unidades de processamento encapsuladas em um único *chip* operando sobre memória compartilhada. A Figura 2.5 apresenta um esquema representativo dessa arquitetura.

Diferentemente das arquiteturas *symmetric multiprocessors*[2] tradicionais na arquitetura Cell existem dois tipos distintos de processadores: uma unidade *PowerPC Processing*

[2]Em arquiteturas *symmetric processors* vários processadores homogêneos acessam uma memória compartilhada com a mesma prioridade de acesso para todos os processadores (NAVAUX; ROSE, 2008).

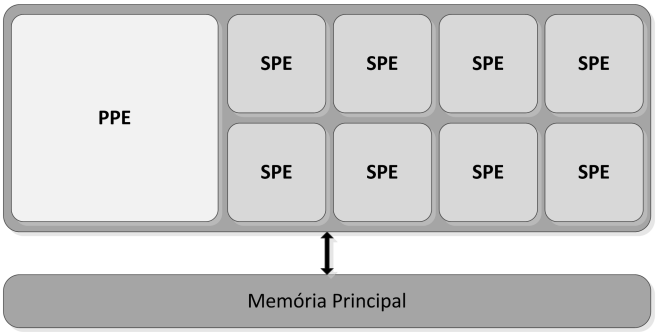


Figura 2.5: Arquitetura Cell BE

Element (PPE) e oito unidades *Synergistic Processor Element* (SPE) (LASTOVETSKY; LASTOVETSKY; DONGARRA, 2009; KORANNE, 2009).

A arquitetura heterogenea Cell foi desenvolvida por um consórcio de empresas inicialmente para o mercado de jogos. Possui foco em aplicações multimídia e de computação intensiva tendo sido utilizada no primeiro computador a romper a barreira de um petaflop de processamento.

Internamente, processadores Cell são constituídos da unidade PPE, das oito unidades SPE e de um barramento interno de alta velocidade para interconexão (EIB - *Element Interconnect Bus*), que realiza a conexão entre PPE, SPEs, controladores de memória e demais componentes do *chip*. A unidade PPE é um processador PowerPC de uso geral enquanto as unidades SPE são direcionadas para operações sobre vetores. As unidades SPE são compostas de um controlador de fluxo de memória (MFC - *memory flow controller*), de uma *synergistic processing unit* (SPU) e de uma pequena memória local.

A programação de aplicações na arquitetura Cell BE é baseada em linguagens como C e C++ e na biblioteca *libspe2*, essa abordagem é apresentada na Seção 3.7 do Capítulo 3.

2.3.2 AMD Fusion

A arquitetura de processadores AMD Fusion combina um ou mais núcleos de processadores x86 de propósito geral com *engines* programáveis para processamento vetorial no mesmo *chip*.

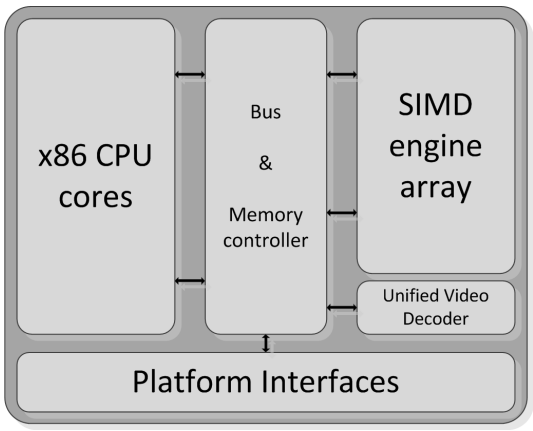


Figura 2.6: Arquitetura AMD Fusion

Internamente, além de núcleos x86 e das *engines* para processamento vetorial (SIMD

- *Single Instruction Multiple Data*) o *chip* também o *Unified Video Decorder* para tarefas de decodificação HD e um barramento de alta velocidade conectando os elementos do presentes no *chip* e este à memória principal, dessa forma a latência da memória é reduzida (STUART; COX; OWENS, 2010; BROOKWOOD, 2010). A Figura 2.6 apresenta um esquema representativo da arquitetura Fusion.

A programação dos núcleos x86 é feita com base em ferramentas tradicionais para a programação de processadores *multicore*. Já a programação das *engines* para processamento vetorial é feita por meio de tecnologias utilizadas para a programação de GPUs como DirectCompute e OpenCL (apresentada no na Seção 3.2 do Capítulo 3).

3 PROGRAMAÇÃO PARALELA EM ARQUITETURAS HÍBRIDAS

Nesse capítulo serão apresentadas ferramentas de programação que oferecem suporte aos recursos de processamento utilizados em arquiteturas híbridas. Essas ferramentas são utilizadas para geração de código a ser executado em aceleradores (p. ex. GPUs).

3.1 CUDA

CUDA (*Compute Unified Device Architecture*) é uma plataforma para programação paralela de propósito geral em GPUs fabricadas pela NVIDIA. A plataforma CUDA pode ser organizada em duas partes: a arquitetura CUDA que é uma abstração do hardware das GPUs NVIDIA e o modelo de programação CUDA que possibilita utilizar a linguagem C para codificar algoritmos a serem executados em GPUs (NVIDIA, 2011b). Na plataforma CUDA a o sistema que comanda a GPU é referenciado como *host* enquanto a GPU é referenciada como *device*.

Em CUDA, um *device* é formado por um conjunto de *Streaming Multiprocessors* que são unidades de processamento que executam independentemente e em paralelo. Os *Streaming Multiprocessors* são constituídos por *Streaming Processors*. Cada *Streaming Processor* é um pequeno núcleo de processamento composto de uma unidade lógica e aritmética para números inteiros (ALU) e uma unidade de números de ponto flutuante (FPU). Todas as unidades *Streaming Processor* de um mesmo *Streaming Multiprocessor* executam simultaneamente a mesma instrução sobre um dado diferente, constituindo um modelo SIMD (*Single Instruction Multiple Data*) (NVIDIA, 2009).

A programação em CUDA é baseada em uma extensão da linguagem C com suporte a alguns recursos da linguagem C++ como *templates* e classes. Essa linguagem é utilizada para codificar um *kernel* CUDA que é função que será compilada para execução em um *streaming processor* na GPU.

Um *kernel* executa em paralelo em um conjunto de *threads* paralelas. O programador ou o compilador CUDA organiza essas *threads* em blocos de *threads* e em *grids* de blocos de *threads*. A GPU instancia um *kernel* em uma *grid* de blocos de *threads*. Cada *thread* de um bloco executa uma instância do *kernel* (NVIDIA, 2009). As *threads* de um bloco podem cooperar entre si por meio de barreiras de sincronização e memória compartilhada. Uma *grid* é um arranjo de blocos de *threads* que podem executar o mesmo *kernel*, lendo entradas e escrevendo resultados na memória global.

Em CUDA, o *host* e os *devices* possuem espaços de memória separados. Isso implica que para executar um *kernel* em um *device*, o programador precisa alocar um espaço na memória global do *device* e transferir os dados da memória do *host* para a memória alo-

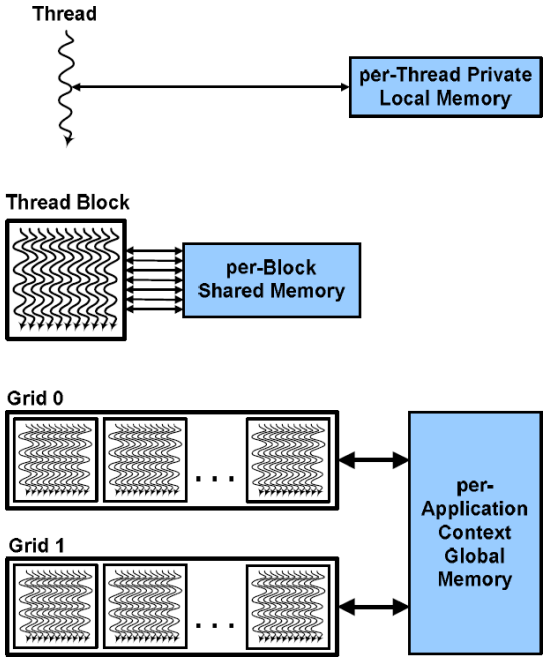


Figura 3.1: Hierarquia de *threads*, blocos e *grids* em CUDA (NVIDIA, 2009)

cada no *device*. Após o final da execução, o programador precosa copiar os resultados da memória do *device* de volta para a memória do *host*. A API CUDA disponibiliza funções como *cudaMalloc(), cudaFree()* e *cudaMemcpy()* para realizar essas ações (KIRK; HWU; HWU, 2010).

A Figura 3.1 representa a hierarquia de memória entre *threads*, blocos e *grids em CUDA*, onde cada *thread* possui um espaço de memória privado por *thread*, cada bloco possui uma memória compartilhada por bloco para comunicação entre as threads e cada *grid* compartilha resultados na memória global da GPU após a sincronização global dos *kernels*.

O Código 3.1 apresenta um exemplo de um *kernel* que pode ser executado em GPUs com suporte à CUDA.

Código 3.1: Exemplo de um *kernel* CUDA (NVIDIA Developer Zone, 2011a)

```
__global__ void matrixMul( float* C, float* A, float* B, int wA, int
    wB){
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];
```

```
        __syncthreads();
#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);
        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

3.2 OpenCL

OpenCL (*Open Computing Language*) (Khronos OpenCL Working Group, 2011) é um padrão para programação paralela em ambientes heterogêneos que possibilita o desenvolvimento de aplicações que executem sobre um conjunto de dispositivos de diferentes fabricantes, como CPUs e GPUs. O OpenCL oferece uma solução para o desenvolvimento de aplicações paralelas de forma independente das ferramentas específicas dos fabricantes.

O padrão especifica uma arquitetura OpenCL que é baseada em uma abstração de baixo nível do hardware. Um dispositivo que ofereça suporte a OpenCL, como determinado modelo de GPU, deve mapear suas características físicas para a abstração proposta pela arquitetura OpenCL (GASTER et al., 2011). A Figura 3.2 ilustra a arquitetura OpenCL.

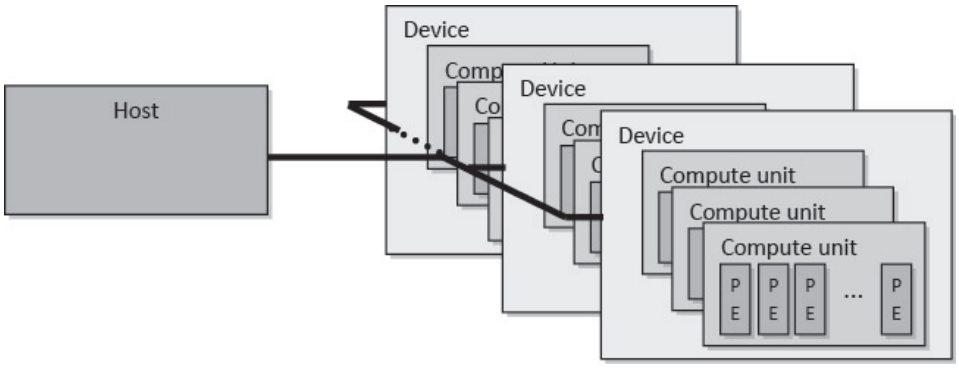


Figura 3.2: Arquitetura OpenCL (GASTER et al., 2011)

Assim como na plataforma CUDA, em OpenCL os espaços de memória do *host* e do *device* são distintos. O OpenCL disponibiliza funções análogas para realizar as alocações e transferências de memória. A Figura 3.3 ilustra o modelo de memória proposto pelo OpenCL. Os tipos de memória *local memory* e a *private memory* nesse modelo equivalem, respectivamente, a *shared memory* e *local memory* na nomenclatura da plataforma CUDA.

O modelo OpenCL possui correspondência em diversos pontos em relação à plataforma CUDA. Por exemplo, os conceitos de *Work item* e *Work group* em OpenCL são análogos aos conceitos de *Thread* e *Block*, respectivamente, em CUDA (KIRK; HWU; HWU, 2010). A semelhança entre os dois modelos também é observada nas APIs como apresentado na tabela 3.1.

A programação em OpenCL é baseada em OpenCL C que é um subconjunto da linguagem C (no padrão C99) com extensões para suportar paralelismo. Essa linguagem é utilizada pra criar *kernels* que executam nos dispositivos OpenCL (Khronos OpenCL

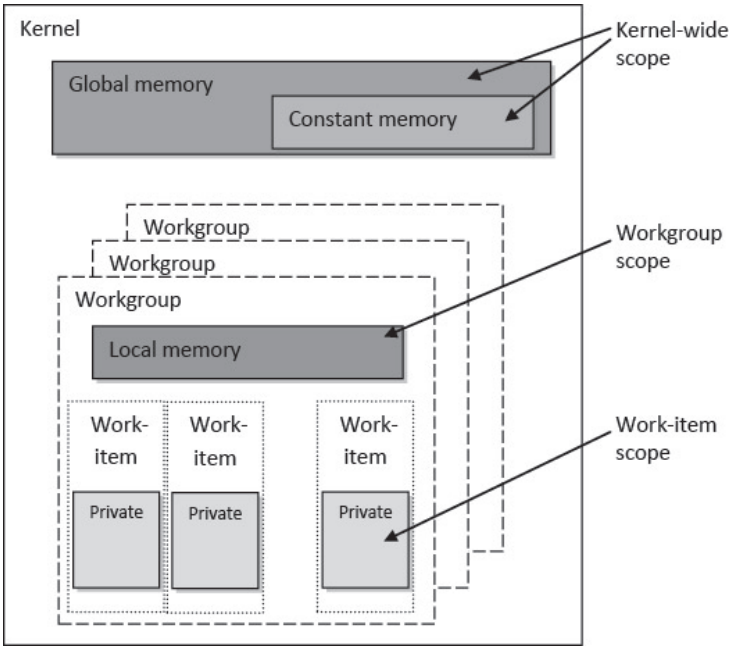


Figura 3.3: Modelo de memória OpenCL (GASTER et al., 2011)

Tabela 3.1: Equivalências entre as APIs de OpenCL e CUDA (KIRK; HWU; HWU, 2010)

OpenCL API	CUDA API
get_global_id(0)	blockIdx.x · blockDim.x + threadIdx.x
get_local_id(0)	threadIdx.x
get_global_size(0)	gridDim.x · blockDim.x
get_local_size(0)	blockDim.x

Working Group, 2011). O Código 3.2 apresenta um exemplo de um *kernel* que pode ser executado em dispositivos com suporte à OpenCL.

Código 3.2: Exemplo de um *kernel* OpenCL (NVIDIA Developer Zone, 2011b)

```
__kernel void matrixMul( __global float* C, __global float* A, __global
    float* B,
     __local float* As, __local float* Bs, int uiWA, int uiWB){
    int bx = get_group_id(0);
    int by = get_group_id(1);
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    int aBegin = uiWA * BLOCK_SIZE * by;
    int aEnd   = aBegin + uiWA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * uiWB;
    float Csub = 0.0f;
    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
        {
        As(ty, tx) = A[a + uiWA * ty + tx];
        Bs(ty, tx) = B[b + uiWB * ty + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
        #pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As(ty, k) * Bs(k, tx);
```

```
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = Csub;
}
```

3.3 Thrust

O Thrust (HOBEROCK; BELL, 2011) é uma biblioteca de algoritmos paralelos para programação em CUDA para GPUs com interface semelhante a STL (*Standard Template Library*) de C++.

A biblioteca oferece uma interface abstrata para algoritmos paralelos básicos como soma de prefixos, ordenação e redução. A transferência de dados entre a memória do sistema e a memória da GPU é facilitada pelo uso de dois *containers vector* que representam a memória do sistema e da GPU, com isso as cópias de dados são feitas por meio de atribuições entre os dois *containers* e a alocação e desalocação de memória é transparente ao programador (HWU, 2011).

O Código 3.3 ilustra um exemplo de uso dos *containers* e de alguns métodos da biblioteca Thrust. Os *containers host_vector* e *device_vector* são utilizados para manipular as memórias do sistema e da GPU, respectivamente. O método *generate* é usado para preencher o *container* com números aleatórios e o método *sort* é usado para ordenar os números armazenados no *container*.

Código 3.3: Exemplo de uso da biblioteca Thrust

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void){
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    thrust::device_vector<int> d_vec = h_vec;
    thrust::sort(d_vec.begin(), d_vec.end());
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
    return 0;
}
```

3.4 HMPP

O HMPP *Hybrid Multicore Parallel Programming Environment* (DOLBEAU; BIHAN; BODIN, 2007; CONSORTIUM, 2011) é um ambiente de programação que oferece suporte à aceleradores por meio de diretivas de compilação. O uso dessas diretivas busca simplificar o uso de aceleradores para aplicações de propósito geral e proporcionar portabilidade de código, uma vez que que o código fonte da aplicação não contém referencias ao *hardware* específico do acelerador. As diretivas de compilação são inseridas em códigos C, C++ e Fortran já existentes para possibilitar geração de código fonte para aceleradores, por exemplo código CUDA e OpenCL.

O HMPP é baseado no conceito de *codelets* que são funções que podem ser executadas

remotamente no *hardware* acelerador. Duas diretivas HMPP são necessárias para obter uma implementação para aceleradores:

- *codelet:* diretiva que declara uma função como um codelet (indica a implementação);
- *callsite:* diretiva inserida antes da chamada da função para especificar a possibilidade do uso do *codelet* (indica a invocação).

O Código 3.4 apresenta um exemplo de uso das diretivas *codelet* e *callsite*. A diretiva *codelet* é utilizada para declarar que a função *matvec* é candidata a ser executada em um acelerador com suporte a CUDA. A partir dessas diretivas é gerado um código CUDA para a função *matvec*. A diretiva *callsite* indica que deve ser feita a alocação do dispositivo CUDA, o envio dos dados, a execução do *codelet* e o recebimento do resultado. Também estão previstas na especificação do modelo diretivas adicionais para transferências de dados visando reduzir o sobrecusto das comunicações (BIHAN et al., 2009).

Código 3.4: Exemplo de uso das diretivas HMPP

```
#pragma hmpp simple codelet, args[outv].io=inout, target=CUDA
static void matvec(int sn, int sm, float inv[sm], float inm[sn][sm],
    float *outv){
  int i, j;
  for (i = 0 ; i < sm ; i++) {
    float temp = outv[i];
    for (j = 0 ; j < sn ; j++) {
      temp += inv[j] * inm[i][ j];
    }
    outv[i] = temp;
  }
}

int main(int argc, char **argv) {
  int n;
  ........
#pragma hmpp simple callsite, args[outv].size={n}
  matvec(n, m, myinc, inm, myoutv);
  ........
}
```

3.5 PGI Accelerator

A ferramenta PGI Accelerator provê diretivas de compilação a serem adicionadas em códigos fonte sequências em linguagem C ou Fortran para paralelização automática de regiões de código a serem executadas em GPUs que ofereçam suporte a CUDA. Um algoritmo adequado para ser paralelizado com essa ferramenta deve conter laços aninhados, nesse caso as iterações podem ser mapeadas para *threads* (GROUP, 2010).

As diretivas de compilação oferecidas pela ferramenta devem ser aplicadas sobre regiões do código sequencial e são organizadas em três tipos:

- diretivas de computação: definem qual região do código do programa será compilada para execução na GPU;

- diretivas de dados: definem quais dados, tipicamente arranjos, devem ser alocados na memória da GPU durante a duração da região de dados, se esses dados devem ser copiados da memória do sistema para a memória da GPU no início da região e/ou copiados da memória da GPU para a do sistema no final da região de dados;

- diretivas de mapeamento de laços: definem que tipo de paralelismo será utilizado para executar o laço da linha seguinte e especificam as variáveis privadas desse laço.

O Código 3.5 apresenta um exemplo de uso das diretivas de compilação. A diretiva *acc region* indica que o bloco de código deve ser compilado para execução na GPU, as opção *copyin* indica quais variáveis de entrada que devem ser copiadas da memória do sistema para a memória da GPU, a opção *copyout* indica a variável de saída que deve copiada da memória da GPU para a memória do sistema. A diretiva *acc for* juntamente com a opção *independent* indica que não existem dependências de dados entre as iterações dos laços e por consequência todos os laços podem ser paralelizados (UNIVERSITY, 2011).

Código 3.5: Exemplo de uso das diretivas PGI Accelerator (UNIVERSITY, 2011)

```
......
double *restrict a;
double *restrict b;
double *restrict c;
......
#pragma acc region copyin(m, n, p, a[0:m*n-1], b[0:n*p-1]) copyout(c[0:
    m*p-1]) local(i, j, k)
{
#pragma acc for independent
    for (j = 0; j < p; j++) {
#pragma acc for independent
        for (i = 0; i < m; i++) {
            c[i*p+j] = 0.0;
#pragma acc for independent
            for (k = 0; k < n; k++) {
                c[i*p+j] += a[i*n+k]*b[k*p+j];
            }
        }
    }
}
......
```

3.6 hiCUDA

A ferramenta hiCUDA (*high-level CUDA*) provê um conjunto de diretivas de compilação a serem aplicadas no código sequencial C/C++ para geração de código CUDA para GPUs. O compilador hiCUDA gera um código fonte em CUDA a partir do código onde foram inseridas as diretivas (HAN; ABDELRAHMAN, 2009).

As diretivas são divididas em dois modelos: modelo de computação e modelo de dados. As diretivas do modelo de dados (*kernel, loop_partition, singular* e *barrier*) permitem que o programador identifique as regiões de código a serem executadas na GPU e como essas devem ser executadas em paralelo. Já as diretivas do modelo de dados (*global, constant, texture* e *shared*) possibilitam ao programador alocar e desalocar memória na GPU e mover dados entre a memória da GPU e a memória do sistema.

O Código 3.6 apresenta um exemplo de uso das diretivas de compilação hiCUDA. Nesse código, as diretivas *global* são usadas para alocar e desalocar os dados na GPU, na alocação, os dados são carregados para a memória da GPU e na desalocação os dados são copiados de volta da memória da GPU para a memória do sistema. As diretivas *kernel* e *kernel_end* delimitam o bloco de código a ser executado na GPU. As diretivas *loop_partition* são utilizadas para dividir as iterações dos laços entre os blocos da GPU (HAN; ABDELRAHMAN, 2011).

Código 3.6: Exemplo de uso das diretivas hiCUDA

```
float A[64][128];
float B[128][32];
float C[64][32];

randomInitArr((float*)A, 64*128);
randomInitArr((float*)B, 128*32);

#pragma hicuda global alloc A[*][*] copyin
#pragma hicuda global alloc B[*][*] copyin
#pragma hicuda global alloc C[*][*]

#pragma hicuda kernel matrixMul tblock(4,2) thread(16,16)
#pragma hicuda loop_partition over_tblock over_thread
for (i = 0; i < 64; ++i) {
#pragma hicuda loop_partition over_tblock over_thread
    for (j = 0; j < 32; ++j) {
        float sum = 0;
        for (kk = 0; kk < 128; kk += 32) {
#pragma hicuda shared alloc A[i][kk:kk+31] copyin
#pragma hicuda shared alloc B[kk:kk+31][j] copyin
#pragma hicuda barrier
            for (k = 0; k < 32; ++k) {
                sum += A[i][kk+k] * B[kk+k][j];
            }
#pragma hicuda barrier
#pragma hicuda shared remove A B
        }
        C[i][j] = sum;
    }
}
#pragma hicuda kernel_end
#pragma hicuda global copyout C[*][*]
#pragma hicuda global free A B C
printMatrix((float*)C, 64, 32);
```

3.7 Programação no processador Cell BE

Devido à organização interna da arquitetura Cell BE (apresentada na Seção 2.3.1 do Capítulo 2) a programação desse processador é dividida em duas partes: a programação do PPE (*PowerPC Processing Element*) e a programação dos SPEs (*Synergistic Processor Elements*). Como o conjunto de instruções dos dois elementos de processamento é diferente são necessários compiladores distintos para cada um dos elementos.

A programação do PPE tem base nas mesmas linguagens utilizadas para programar a arquitetura PowerPC como Fortran, C e C++. Além disso, como o PPE comanda as unidades SPE, é necessária uma biblioteca adicional, chamada *libspe2*, para possibilitar

o gerenciamento destes. A *libspe2* ou *SPE Runtime Management Library* (IBM, 2006) é uma API de baixo nível que oferece mecanismos para a aplicação acessar os SPEs. Esses mecanismos possibilitam criar, iniciar e parar contextos SPE, manipular as transferências de dados entre o PPE e os SPEs (via DMA) e carregar os executáveis SPE para memória local destes (IBM; SONY; TOSHIBA, 2008).

Os SPEs também são programados com linguagens como Fortran, C e C++ porém o suporte a essas linguagens é restrito e muitas funcionalidades comuns não estão disponíveis devido as limitações do *hardware* do SPE como a pouca capacidade da memória local. Apesar das restrições algumas extensões foram incluídas para oferecer suporte à instruções vetoriais/SIMD (IBM; SONY; TOSHIBA, 2008).

Existem dois compiladores que oferecem suporte aos conjuntos de instruções PPE e SPE da arquitetura Cell: o GCC e o IBM XL. A geração de código executável para a arquitetura envolve compilar o código fonte SPE e ligá-lo ao executável SPE, este é incluído no arquivo objeto PPE que é ligado com o restante do código PPE para produzir um executável Cell BE.

4 AMBIENTES DE EXECUÇÃO PARA ARQUITETURAS HÍBRIDAS

Nesse capítulo são abordados ambientes de execução com suporte a arquiteturas paralelas híbridas. Usualmente, cada tecnologia de processamento possui seus próprios mecanismos para gerenciar a execução (p. ex CUDA para GPUs NVIDIA) e a manipulação dos dados (p. ex DMA na arquitetura Cell BE). Os ambientes de execução para arquiteturas híbridas têm por finalidade facilitar a execução de aplicações paralelas em sistemas compostos de vários recursos de processamento distintos. Tais ambientes possibilitam o aproveitamento eficiente dos recursos distintos por meio de algoritmos de escalonamento e balanceamento de carga e do gerenciamento das alocações e transferências de dados entre as memórias dos recursos de processamento.

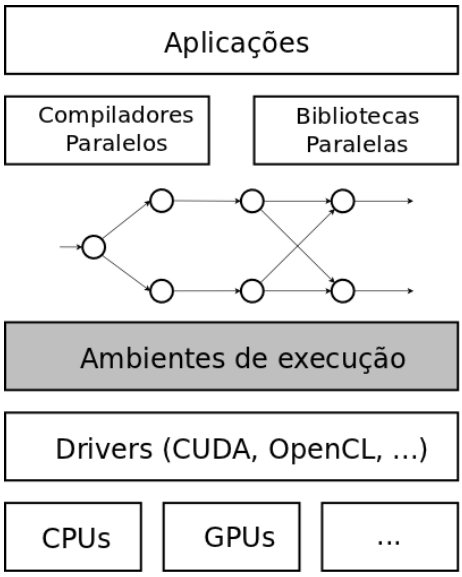


Figura 4.1: Ambientes de execução em uma arquitetura paralela híbrida

4.1 StarPU

O StarPU (AUGONNET et al., 2009) é um ambiente de execução que oferece suporte para arquiteturas *multicore* híbridas. O ambiente reúne uma abordagem unificada dos recursos de processamento, como CPUs *multicore* e aceleradores simultaneamente, mecanismos para escalonar de forma eficiente as tarefas na arquitetura híbrida e transferências de dados de forma transparente e portável entre os recursos de processamento da

arquitetura.

O StarPU está organizado em três componentes principais (AUGONNET; THIBAULT; NAMYST, 2010):

- Gerenciamento de dados: uma biblioteca de alto nível que automatiza de forma eficiente as transferências de dados entre os recursos de processamento heterogêneos;
- Modelo de execução unificado: uma abordagem uniforme para paralelismo de dados e tarefas em plataformas híbridas;
- Políticas de escalonamento: um *framework* que permite projetar políticas de escalonamento a serem usadas pelo StarPU.

Usualmente, os processadores tradicionais (CPUs) e os aceleradores não podem acessar de forma transparente a memória um do outro, logo executar aplicações simultaneamente nesses recursos distintos implica em mover explicitamente os dados entre os vários recursos existentes, o StarPU oferece uma biblioteca para automatizar essas transferências. Essa biblioteca implementa uma memória compartilhada virtual via software utilizando uma consistência de memória relaxada e recursos de replicação de dados. Todas as tarefas possuem referencias explicitas aos seus dados de entrada e saída, com isso o escalonador pode buscar os dados antes que a tarefa comece a executar e escrever os dados quando necessário (AUGONNET; NAMYST, 2009).

O modelo de execução do StarPU propõe uma abordagem de tarefas independente da arquitetura base. São definidos *codelets* como uma abstração de uma tarefa que pode ser executada em um núcleo de uma CPU *multicore* ou submetido a um acelerador. Cada *codelet* pode ter múltiplas implementações, uma para cada arquitetura em que o *codelet* pode ser executado. Cada implementação utiliza as linguagens de programação ou bibliotecas específicas para a arquitetura alvo. Um *codelet* contém uma descrição dos dados e o tipo de acesso (leitura, escrita ou ambos). *Codelets* e são lançados de forma assíncrona. Com isso o escalonador pode reordenar as tarefas para melhorar o desempenho respeitando as dependências entre elas. Uma aplicação StarPU é descrita como um conjunto de *codelets* com suas dependências de dados. O Código 4.1 apresenta um exemplo de declaração de um *codelet*, são informados os recursos de processamento para os quais o *codelet* possui implementação e as funções que possuem essas implementações. Parâmetros adicionais como o modelo de desempenho também podem ser informados. O Código 4.2 ilustra um exemplo de submissão de uma tarefa a partir de um *codelet*.

Código 4.1: Exemplo de declaração de um codelet StarPU

```
void scal_gpu ( void *buffers[], void *cl_arg ) {
    /* CUDA code*/
    ...
}
void scal_cpu ( void *buffers[], void *cl_arg ) {
    /* CPU code */
    ...
}
static starpu_codelet scal_cl = {
    .where = STARPU_CPU|STARPU_CUDA,
    .cpu_func = scal_cpu,
    .cuda_func = scal_gpu,
    .model = &starpu_scal_model,
    .nbuffers = N_BUFFERS
}
```

Código 4.2: Exemplo de submissão de uma tarefa StarPU

```
...
struct starpu_task *task = starpu_task_create();

task->cl = &scal_cl;
task->buffers[0].handle = vector_handle;
task->buffers[0].mode = STARPU_RW;
task->cl_arg = &factor;
task->cl_arg_size = sizeof(factor);

starpu_task_submit(task);
starpu_task_wait_for_all();
...
```

O *framework* para políticas de escalonamento permite especificar como as tarefas serão distribuídas e redistribuídas entre os recursos de processamento. As políticas de escalonamento influenciam diretamente no desempenho da aplicação por meio da distribuição e redistribuição das tarefas de forma eficiente, visando manter a carga de trabalho dos diferentes recursos balanceada ou favorecendo a localidade dos dados proporcionando melhor uso de memórias *cache* por exemplo. Novas políticas de escalonamento para o StarPU podem ser implementadas utilizando o *framework*. O StarPU inclui um conjunto de políticas de escalonamento já implementadas (TEAM, 2011):

- *eager:* é um escalonador guloso baseado em uma fila de tarefas centralizada, da qual os recursos de processamento (*workers*) buscam tarefas para executar;

- *prio:* semelhante ao escalonador eager porém a fila de tarefas é mantida ordenada de acordo com a prioridade associada a tarefa. Cada tarefa pode ter prioridade entre -5 (menor prioridade) e $+5$ (maior prioridade), a prioridade por padrão é 0;

- *random:* distribui as tarefas aleatoriamente entre os *workers*de acordo com o desempenho assumido por cada *worker*;

- *ws(work stealing):* é baseado em roubo de tarefas. As tarefas são escalonadas no próprio *worker*. Quando um *worker*fica ocioso, ele rouba uma tarefa do *worker*mais carregado;

- *dm (deque model):* utiliza modelos de desempenho para as tarefas, escalona as tarefas de forma a minimizar o tempo de execução de cada tarefa;

- *dmda (deque model data aware):* semelhante ao dm porém também considera o tempo de transferência dos dados;

- *dmdar (deque model data aware ready):* semelhante ao dm e dmda porém ordena as tarefas na fila de cada *worker*pelo número de *buffers* de dados disponíveis;

- *dmdas (deque model data aware sorted):* semelhante ao dm e dmda porém suporta valores de prioridade arbitrários;

- *heft:* semelhante ao dm e dmda mas também suporta pacotes de tarefas.

4.2 XKaapi

O XKaapi (INRIA; MOAIS; LIG, 2011) é uma reimplementação do Kaapi com suporte a paralelismo de tarefas de grão fino. O Kaapi (GAUTIER; BESSERON; PIGEON, 2007) (*Kernel for Adaptative, Asynchronous Parallel and Interactive programming*) é um ambiente de execução para computação paralela em arquiteturas CPU *multicore* e *clusters*. A implementação atual do XKaapi oferece suporte à arquiteturas *multicore*. Uma extensão para incluir suporte à GPUs foi proposta em (HERMANN et al., 2010) e está disponível nas versões em desenvolvimento do XKaapi.

O ambiente de execução XKaapi visa simplificar o desenvolvimento de aplicações paralelas provendo uma abstração da arquitetura do sistema e balanceamento de carga dinâmico e automático por meio de algoritmos de roubo de tarefas (*work stealing*).

O XKaapi é composto pelo *kernel* por um conjunto de APIs. O *kernel*, escrito em C, é o ambiente de execução para as APIs e oferece escalonamento baseado em roubo de tarefas. O conjunto de APIs possui suporte à diferentes modelos de programação. A programação pode ser feita utilizando a interface de baixo nível em C ou através de uma das APIs disponíveis:

- Athapascan: interface obsoleta do Kaapi baseada em um DFG[3] (*Data Flow Graph*) para C++ ;
- Kaapi++: interface atual do XKaapi baseada em um DFG para C++;
- KaSTL: biblioteca semelhante a STL de C++ porém paralela e implementada sobre o Kaapi;
- AAPI: interface para algoritmos adaptativos;
- KaFOR: interface do Kaapi para Fortran.

O escalonamento das tarefas é feito a partir de algoritmos básicos como *work stealing* dinâmico e particionamento estático de tarefas a partir do grafo de dependência de dados. A abordagem estática é voltada para aplicações de computação numérica iterativa enquanto o *work stealing* possui melhor desempenho em aplicações recursivas (GAUTIER; BESSERON; PIGEON, 2007).

A interface Kaapi++ é API do XKaapi para C++. Essa interface possui três componentes principais:

- Assinatura da tarefa (*task signature*): define o número de parâmetros da tarefa, o tipo e o modo de acesso de cada um desses parâmetros. Os modos de acesso podem ser leitura (*«R»*), escrita (*«W»*), leitura e escrita (*«RW»*) e leitura cumulativa (*«CW»*);
- Implementação da tarefa (*task implementation)*: especifica a implementação da tarefa para uma dada arquitetura;
- Criação da tarefa (*task creation*): submete a tarefa para a pilha de execução;

A execução de uma tarefa inicia somente quando todos os seus parâmetros de entrada foram produzidos, ao final da execução da aplicação todas as tarefas criadas devem ter sido executadas. Os parâmetros podem ser passados por cópia ou por referencia, nesse

[3] O grafo de fluxo de dados (DFG) representa as dependências entre tarefas e dados

caso o modo de acesso deve ser informado. Um dado é compartilhado entre duas tarefas se e somente se ambas possuírem o mesmo ponteiro nos parâmetros efetivos. O Código 4.3 ilustra um exemplo das etapas de assinatura, implementação e criação de tarefas com a API Kaapi++.

Código 4.3: Exemplo de declaração de tarefas com Kaapi++

```
struct TaskHello : public ka::Task<2>::Signature<std::string, double>
    {};

template<> struct TaskBodyCPU<TaskHello> {
  void operator() ( std::string msg, double n ){
    std::cout << "Hello World !, msg=" << msg << ", n=" << n << std::
        endl;
  }
};

struct doit {
  void operator()(int argc, char** argv ){
    double n = 3.1415;
    if (argc >1)
        n = atof(argv[1]);
    ka::Spawn<TaskHello>()( "zzz", n );
  }
};
```

Uma extensão a interface Kaapi++ para oferecer suporte à GPUs por meio de múltiplas implementações para cada tarefa foi proposta em (HERMANN et al., 2010). Esse suporte está disponível nas versões de teste do XKaapi. Com essa abordagem a etapa de *task implementation* pode possuir mais de uma implementação para uma determinada tarefa. Todas as implementações fornecidas devem respeitar a declaração de *task signature* e pelo menos uma implementação de cada tarefa deve ser fornecida. Quando mais de uma implementação está disponível o escalonador do XKaapi decide em tempo de execução qual das implementações/arquitetura será utilizada. O Código 4.4 ilustra um exemplo de uso de tarefas com múltiplas implementações no Kaapi++ (LIMA; MAILLARD, 2011).

Código 4.4: Exemplo de declaração de tarefas com múltiplas implementações no Kaapi++

```
struct TaskHello public ka::Task<1>::Signature<int>{} ;

template<> struct TaskBodyCPU<TaskHello>{
    void operator ( ) (int n){
        /* CPU Implementation ... */
    }
};

template<> struct TaskBodyGPU<TaskHello> {
    void operator ( ) (int n){
        /* GPU Implementation ... */
    }
};

    ...
    ka::Spawn<TaskHello>(n)
    ...
```

Recentemente foi apresentada uma nova interface (LE MENTEC; GAUTIER; DANJEAN, 2011) baseada em diretivas de compilação adicionadas a códigos sequenciais. As

diretivas são inseridas para identificar funções a serem transformadas em tarefas. Na diretiva são incluídas especificações quanto ao modo de acesso na memória (leitura, escrita, acesso exclusivo e redução) de cada parâmetro da função. Com isso, o compilador insere as chamadas ao ambiente de execução para a criação das tarefas e durante a execução as dependências são detectadas e as tarefas são escalonadas. O Código 4.5 apresenta um exemplo de uso da interface de diretivas de compilação do XKaapi.

Código 4.5: Exemplo de uso das diretivas de compilação XKaapi

```
#pragma kaapi task write(buffer[size]) value(size) read(msg)
void write_msg(int size, char* buffer, const char* msg){
  snprintf(buffer, size, "%s", msg);
}

#pragma kaapi task read(msg)
void print_msg(const char* msg){
  printf("%s\n", msg);
}

int main(int argc, char** argv){
  char buffer[32];
#pragma kaapi parallel
  {
    write_msg(32,buffer, "This is may be a too long message for the
       buffer.");
    print_msg(buffer);
  }
  return 0;
}
```

4.3 StarSs

O StarSs (PLANAS et al., 2009; MEENDERINCK; JUURLINK, 2010) é um ambiente de execução baseado na adição de diretivas ao código sequencial C/Fortran para identificar tarefas (trechos de código que podem ser executados em paralelo) e os argumentos de entrada e saída dessas tarefas. O Código 4.6 apresenta a estrutura das diretivas de compilação propostas pelo StarSS.

Código 4.6: Estrutura das diretivas de compilação StarSs

```
#pragma css task [input ( parameters ) ] \
    [output ( parameters ) ] \
    [inout ( parameters )] \
    [target device( [cell, smp, cuda] ) ] \
    [implements ( task_name ) ] \
    [reduction ( parameters ) ] \
    [ highpriority ]
```

O StarSs oferece suporte a diversas arquiteturas por meio de extensões específicas:

- SMPSs para arquiteturas *multicore* e multiprocessador homogêneas;
- CellSs para arquiteturas baseadas em processadores Cell BE;
- GPUSs para arquiteturas compostas por GPUs NVidia.

O ambiente de execução que é responsável pela paralelização automática em tempo de execução. Esse ambiente de execução identifica quais tarefas podem ser executadas em paralelo com base nos atributos de entrada e saída e as escalona entre os recursos de processamento que compõem o sistema. Isso é feito através de um grafo de dependência de dados construído em tempo de execução. Além disso, o ambiente de execução realiza o balanceamento da carga e otimizações com base na localidade dos dados. Nas implementações CellSs e GPUSs a migração de dados de/para o dispositivo é feita de forma transparente.

O suporte a plataformas heterogêneas simultaneamente foi proposto na extensão GPUSs (AYGUADé et al., 2009). Além do suporte ao hardware de GPUs foram incorporados novos parâmetros nas diretivas de compilação de forma a permitir que as tarefas tenham mais de uma implementação (uma por dispositivo suportado).

O ambiente de execução é estruturado em:

- *master thread*: gera as tarefas e as insere no grafo de dependência;
- *helper thread*: consome as tarefas que estão no grafo de dependência e as mapeia ao dispositivo mais adequado;
- *worker threads*: aguardam por tarefas disponíveis, realizam as transferências de dados entre a memória da GPU e a memória principal e realizam as chamadas para as tarefas na GPU.

O escalonador do ambiente de execução é uma variação da técnica de *work pushing* pois as tarefas são geradas pela *master thread* enquanto a *helper thread* insere as tarefas nas filas das *worker threads*.

O Código 4.7 apresenta um exemplo das diretivas propostas no GPUSs para suporte a tarefas com múltiplas implementações.

Código 4.7: Exemplo de tarefas com múltiplas implementações no StarSs/GPUSs

```
#pragma css task inout(A[NT][NT])
void chol_spotrf (float *A);

#pragma css task inout(A[NT][NT]) target device (cuda) implements (
    chol_spotrf)
void chol_spotrf_cuda(float *A){
    // CUDA GPU Kernel code
}

#pragma css task inout(A[NT][NT]) target device (smp) implements (
    chol_spotrf)
void chol_spotrf_smp(float *A){
    // SMP code
}
```

5 CONSIDERAÇÕES FINAIS

Neste trabalho, apresentou-se uma visão geral sobre a programação e gerenciamento da execução de aplicações paralelas em arquiteturas híbridas. No Capítulo 2 foram apresentados os recursos de processamento que compõe as arquiteturas paralelas híbridas atuais.

No Capítulo 3, foram apresentadas ferramentas que possibilitam a programação dos recursos de processamento auxiliares, como GPUs. Entre as ferramentas apresentadas, CUDA3.1 e OpenCL3.2 estão situadas no nível mais próximo do *hardware*, o que exige conhecimento sobre a arquitetura e o funcionamento do recurso alvo. A biblioteca Thrust3.3 oferece alguma abstração do *hardware* em relação a CUDA e OpenCL através de um conjunto de *templates*. Já as ferramentas hiCUDA3.6, HMPP3.4 e PGI Accelerator3.5 são baseadas na adição de diretivas de compilação sobre códigos sequenciais. Entretanto, a ferramenta hiCUDA possui menor nível de abstração pois são necessárias muitas diretivas e estas são muito parecidas aos comandos CUDA originais, o que implica que programador conheça CUDA para utilizar o hiCUDA. Já o HMPP e o PGI Accelerator oferecem uma abstração de alto nível, sendo necessário um número pequeno de diretivas. Porém, ambas são ferramentas proprietárias e a documentação é pouco detalhada sobre o funcionamento e implementação das ferramentas. A programação para o processador Cell BE é baseada em linguagens como C e C++ associadas uma biblioteca de baixo nível chamada *libspe2*.

O Capítulo 4 apresentou os ambientes de execução StarPU4.1, XKaapi4.2 e StarSs4.3 que oferecem suporte para arquiteturas híbridas. As principais funcionalidades desses ambientes são o escalonamento das tarefas entre os diferentes recursos que compõe a arquitetura e o gerenciamento dos dados. Os três ambientes escalonam as tarefas com base em grafos de dependências de dados gerados em tempo de execução.

Portanto, podemos visualizar a programação de arquiteturas paralelas híbridas como a união de dois tipos de ferramentas: as que possibilitam a programação de recursos de processamentos auxiliares como aceleradores e as ferramentas que oferecem suporte em tempo de execução, facilitando o escalonamento e a manipulação dos dados entre os recursos de processamento.

REFERÊNCIAS

ASANOVIC, K. et al. A view of the parallel computing landscape. **Commun. ACM**, New York, NY, USA, v.52, p.56–67, Oct. 2009.

AUGONNET, C. et al. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: INTERNATIONAL EURO-PAR CONFERENCE ON PARALLEL PROCESSING, 15., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2009. p.863–874. (Euro-Par '09).

AUGONNET, C.; NAMYST, R. A Unified Runtime System for Heterogeneous Multicore Architectures. In: CÉSAR, E. et al. (Ed.). **Euro-Par 2008 Workshops - Parallel Processing**. Berlin, Heidelberg: Springer-Verlag, 2009. p.174–183.

AUGONNET, C.; THIBAULT, S.; NAMYST, R. Automatic calibration of performance models on heterogeneous multicore architectures. In: PARALLEL PROCESSING, 2009., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.56–65. (Euro-Par'09).

AYGUADé, E. et al. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: SIPS, H.; EPEMA, D.; LIN, H.-X. (Ed.). **Euro-Par 2009 Parallel Processing**. [S.l.]: Springer Berlin / Heidelberg, 2009. p.851–862. (Lecture Notes in Computer Science, v.5704).

BIHAN, S. et al. Directive-based Heterogeneous Programming A GPU-Accelerated RTM Use Case. In: INTERNATIONAL CONFERENCE ON COMPUTING, COMMUNICATIONS AND CONTROL TECHNOLOGIES: CCCT 2009, 7. **Anais...** [S.l.: s.n.], 2009.

BLUMOFE, R. Cilk: an efficient multithreaded runtime system. **Journal of Parallel and Distributed Computing**, [S.l.], v.37, n.1, p.55–69, 1996.

BLUMOFE, R. D. et al. Cilk: an efficient multithreaded runtime system. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, New York, NY, USA. **Proceedings...** ACM, 1995. p.207–216. (PPOPP '95).

BROOKWOOD, N. **AMD Fusion^{TM}Family of APUs**: enabling a superior, immersive pc experience. 2010.

CHAPMAN, B.; JOST, G.; PAS, R. **Using OpenMP**: portable shared memory parallel programming. [S.l.]: MIT Press, 2007. n.v. 10. (Scientific and engineering computation).

CONSORTIUM, O. **OpenHMPP Concepts & Directives**. [S.l.: s.n.], 2011.

DOLBEAU, R.; BIHAN, S.; BODIN, F. HMPP: a hybrid multi-core parallel programming environment. In: FIRST WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS. **Anais...** [S.l.: s.n.], 2007.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN 1998 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION - PLDI '98, New York, New York, USA. **Proceedings...** ACM Press, 1998. p.212–223.

GASTER, B. et al. **Heterogeneous Computing with OpenCL**. [S.l.]: Elsevier Science, 2011.

GAUTIER, T.; BESSERON, X.; PIGEON, L. KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PARALLEL SYMBOLIC COMPUTATION, 2007., New York, NY, USA. **Proceedings...** ACM, 2007. p.15–23. (PASCO '07).

GOKHALE, M.; GRAHAM, P. **Reconfigurable computing**: accelerating computation with field-programmable gate arrays. [S.l.]: Springer, 2005.

GROUP, T. P. **PGI Accelerator Programming Model for Fortran & C**. [S.l.: s.n.], 2010.

HAN, T. D.; ABDELRAHMAN, T. S. hiCUDA: a high-level directive-based language for gpu programming. In: WORKSHOP ON GENERAL PURPOSE PROCESSING ON GRAPHICS PROCESSING UNITS - GPGPU-2, 2., New York, New York, USA. **Proceedings...** ACM Press, 2009. p.52–61.

HAN, T. D.; ABDELRAHMAN, T. S. hiCUDA: high-level gpgpu programming. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.22, n.1, p.78–90, Jan. 2011.

HERMANN, E. et al. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: EURO-PAR CONFERENCE ON PARALLEL PROCESSING: PART II, 16., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2010. p.235–246. (Euro-Par'10).

HOBEROCK, J.; BELL, N. **Thrust**: a parallel template library. Version 1.4.0.

HWU, W.-M. W. Thrust - A Productivity-Oriented Library for CUDA. In: **GPU Computing Gems**: jade edition. [S.l.]: Elsevier, 2011. p.359–373.

IBM. **SPE Runtime Management Library**. [S.l.: s.n.], 2006.

IBM; SONY; TOSHIBA. **Cell Broadband Engine Programming Handbook**. [S.l.: s.n.], 2008.

INRIA; MOAIS; LIG. **XKAAPI - Kernel for Adaptative, Asynchronous Parallel and Interactive programming**. Disponível em http://kaapi.gforge.inria.fr/. Acesso em Novembro de 2011.

Khronos OpenCL Working Group. **The OpenCL Specification (version 1.2)**. 2011. 377p.

KIRK, D.; HWU, W.; HWU, W. **Programming massively parallel processors**: a hands-on approach. [S.l.]: Morgan Kaufmann Publishers, 2010. (Applications of GPU Computing Series).

KORANNE, S. **Practical Computing on the Cell Broadband Engine**. [S.l.]: Springer, 2009.

LASTOVETSKY, A.; LASTOVETSKY, A.; DONGARRA, J. **High-performance heterogeneous computing**. [S.l.]: Wiley, 2009. (Wiley series on parallel and distributed computing).

LE MENTEC, F.; GAUTIER, T.; DANJEAN, V. **The X-Kaapi's Application Programming Interface. Part I: data flow programming**. [S.l.]: INRIA, 2011. Rapport Technique. (RT-0418).

LIMA, J.; MAILLARD, N. Linear Algebra Algorithms for Hybrid Architectures with XKaapi. In: WSPPD 2011 - IX WORKSHOP DE PROCESSAMENTO PARALELO E DISTRIBUÍDO, Porto Alegre. **Anais...** [S.l.: s.n.], 2011.

MEENDERINCK, C.; JUURLINK, B. A Case for Hardware Task Management Support for the StarSS Programming Model. In: EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN: ARCHITECTURES, METHODS AND TOOLS, 2010. **Anais...** IEEE, 2010. p.347–354.

NAVAUX, P.; ROSE, C. **ARQUITETURAS PARALELAS**. [S.l.]: BOOKMAN COMPANHIA ED, 2008.

NICKOLLS, J.; DALLY, W. J. The GPU Computing Era. **IEEE Micro**, [S.l.], v.30, n.2, p.56–69, Mar. 2010.

NVIDIA. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture. **ReVision**, [S.l.], p.1–22, 2009.

NVIDIA. **CUDA Programming Guide**. [S.l.: s.n.], 2011.

NVIDIA. **CUDA C Programming Guide**. [S.l.: s.n.], 2011.

NVIDIA Developer Zone. **CUDA C/C++ SDK CODE Samples**. Disponível em http://developer.nvidia.com/cuda-cc-sdk-code-samples. Acesso em Outubro de 2011.

NVIDIA Developer Zone. **OpenCL SDK Code Samples**. Disponível em http://developer.nvidia.com/opencl-sdk-code-samples. Acesso em Outubro de 2011.

OWENS, J. et al. GPU Computing. **Proceedings of the IEEE**, [S.l.], v.96, n.5, p.879 –899, may 2008.

PLANAS, J. et al. Hierarchical Task-Based Programming With StarSs. **Int. J. High Perform. Comput. Appl.**, Thousand Oaks, CA, USA, v.23, p.284–299, August 2009.

RAUBER, T.; RÜNGER, G. **Parallel Programming**: for multicore and cluster systems. [S.l.]: Springer, 2010.

REINDERS, J. **Intel threading building blocks**: outfitting c++ for multi-core processor parallelism. [S.l.]: O'Reilly, 2007. (O'Reilly Series).

SANDERS, J.; KANDROT, E. **CUDA by Example**: an introduction to general-purpose gpu programming. [S.l.]: Pearson Education, 2010.

SHAN, A. Heterogeneous Processing: a strategy for augmenting moore's law. **Linux Journal**, Seattle, WA, USA, v.2006, n.142, p.7, 2006.

STUART, J. A.; COX, M.; OWENS, J. D. GPU-to-CPU Callbacks. In: UCHPC 2010: PROCEEDINGS OF THE THIRD WORKSHOP ON UNCONVENTIONAL HIGH PERFORMANCE COMPUTING (EURO-PAR 2010 WORKSHOPS). **Anais...** Springer, 2010. (UCHPC 2010: Proceedings of the Third Workshop on UnConventional High Performance Computing (Euro-Par 2010 Workshops)).

TEAM, I. R. **StarPU Handbook**. Disponível em http://runtime.bordeaux.inria.fr/StarPU/starpu.html. Acesso em Novembro de 2011.

UNIVERSITY, N. C. S. **Using PGI Compilers to do GPU Computations**. Disponível em http://www.ncsu.edu/itd/hpc/Documents/BladeCenter/PGI-gpu.php. Acesso em Novembro de 2011.