Capítulo

7

Introdução à Computação Heterogênea

Denise Stringhini, Rogério A. Gonçalves, Alfredo Goldman

Resumo

Diversos tipos de coprocessadores tem sido utilizados a fim de acelerar a execução de aplicações que executam em um nó computacional. Estes coprocessadores tem sido chamados de aceleradores e são responsáveis pela natureza heterogênea destes nós computacionais. Este tutorial introduz conceitos básicos de computação heterogênea a partir de uma das opções mais populares entre os aceleradores, que são as GPUs. Num primeiro momento é apresentada uma contextualização, onde são descritos os principais componentes de sua arquitetura, relacionando-os com os modelos SIMD, SPMD e SIMT, largamente explorados em computação heterogênea. Em seguida, o tutorial apresenta as alternativas de ambientes e ferramentas de programação para estes aceleradores, juntamente com exemplos de programação heterogênea que unem CPU e GPU. Neste contexto, são apresentados ambientes de programação tais como CUDA, OpenCL, OpenMP e OpenACC. Eles podem ser usados para se obter paralelismo em arquiteturas heterogêneas compostas de CPU e GPU.

Abstract

Several types of coprocessors have been used in order to accelerate the execution of one node only applications. These coprocessors have been named accelerators and are responsible for the heterogeneous nature of these computational nodes. This tutorial introduces heterogeneous computing basic concepts from one the most popular accelerators options that are the GPUs. First, a contextualization is presented where its main architecture components are described and related to the SIMD, SPMD and SIMT models, widely explored in heterogeneous computing. After that, the tutorial presents the alternatives in programming environments and tools for these accelerators along with heterogeneous programming examples that unify CPU and GPU. In this context, programming environments as CUDA, OpenCL, OpenMP and OpenACC are presented. They could be used in order to obtain parallelism in heterogeneous architectures composed by CPU and GPU.

7.1. Introdução

O termo "computação heterogênea" tem sido empregado para designar o uso de diferentes arquiteturas em um mesmo nó computacional a fim de se obter melhores desempenhos das aplicações que executarão neste nó. Entre as alternativas destacam-se arquiteturas como GPUs (*Graphics Processing Units*) e FPGAs (*Field-programmable Gate Arrays*), ambos também conhecidos como aceleradores.

Devido aos ganhos de desempenho recentemente observados em aplicações que utilizam tais aceleradores, assim como seu baixo consumo de energia, fabricantes como Intel, AMD e ARM tem anunciado projetos de chips heterogêneos. Neste contexto, é necessário um bom conhecimento dos modelos de programação paralela empregados nessas arquiteturas. Em geral, os modelos de programação dos aceleradores correspondem a uma evolução do modelo SIMD, onde uma mesma instrução é aplicada sobre um conjunto de dados em paralelo. Em GPUs, por exemplo, um mesmo trecho de código, chamado *kernel*, é replicado em até milhares de *threads* que executam de forma concorrente sobre um grande conjunto de dados.

O principal objetivo deste tutorial é justamente apresentar diferentes ambientes e ferramentas de programação que permitem que um código sendo executado na CPU possa enviar trabalho para um acelerador. A heterogeneidade deste tipo de computação está justamente em se desenvolver aplicações que explorem todos os dispositivos presentes na máquina. Além disso, é importante salientar que o modelo de programação dos aceleradores não é adequado a todo o tipo de algoritmo. Assim, é importante que o desenvolvedor saiba identificar as tarefas mais adequadas para a execução num acelerador e aquelas mais adequadas à execução numa CPU tradicional. O estudo das arquiteturas dos aceleradores, assim como dos ambientes de desenvolvimento disponíveis pode auxiliar os desenvolvedores a ter esta percepção.

Este tutorial apresenta os principais ambientes e ferramentas disponíveis para a computação heterogênea com aceleradores do tipo GPU, juntamente com exemplos introdutórios. A segunda seção apresenta brevemente a nomenclatura relacionada ao paralelismo que é utilizada no restante do texto. A terceira seção tem a função de contextualizar as GPUs, foco deste tutorial, entre as principais arquiteturas heterogêneas em utilização no momento. A quarta seção apresenta as arquiteturas das GPUs mais atuais da NVIDIA, a Fermi e a Kepler, que serão os modelos de aceleradores utilizados como principal exemplo neste tutorial. A quinta seção apresenta o OpenMP que permite utilizar *threads* na CPU e pode ser utilizado em conjunto com os ambientes de programação para aceleradores. A sexta seção aborda a plataforma de programação CUDA para GPUs, enquanto a sétima apresenta a alternativa OpenCL, com uma abordagem mais ampla (heterogênea) que pretende abranger uma série de dispositivos, principalmente CPUs e GPUs. A oitava seção apresenta o OpenACC que possui um conjunto de primitivas semelhantes ao OpenMP, porém com utilização em GPUs. Dado que a grande maioria destas ferramentas e ambientes são apresentados para a linguagem C, a última seção apresenta algumas alternativas para que se possa utilizar aceleradores do tipo GPU em outras linguagens (Java, Python e C++/STL).

7.2. Conceitos de paralelismo

O objetivo desta seção é apresentar os principais conceitos de paralelismo relacionados à computação heterogênea. Neste sentido, a terminologia utilizada em HPC (*High Performance Computing*) ou PAD no português (Processamento de Alto Desempenho) é apresentada a fim de familiarizar os leitores/audiência com os termos a serem usados ao longo do capítulo.

7.2.1. Processamento de Alto Desempenho

A busca por alto desempenho de aplicações fez com que as melhorias se expandissem para domínios externos aos de arquiteturas convencionais. Os *clusters* de computadores combinados e interconectados com uma rede de alta velocidade, multiprocessadores de memória compartilhada (SMP) conectando múltiplos processadores em um único sistema de memória e os *Chip Multi-Processor* (CMP) contendo múltiplos núcleos integrados no mesmo chip, são exemplos de máquinas paralelas.

É notável que as máquinas mais rápidas do mundo hoje em dia, relacionadas no site TOP500 (www.top500.org), possuem paralelismo em todos os níveis de suas arquiteturas. Embora praticamente todas as máquinas relacionadas sejam clusters ou MPPs (*Massively Parallel Processors*), compostos de centenas de nós de processamento, estes nós são cada vez mais heterogêneos, onde o paralelismo pode ser explorado de diferentes maneiras.

Segundo a lista TOP500 de novembro de 2011, o computador mais rápido do mundo, o computador K do Japão, conseguiu um desempenho de 10.51 PetaFlop/s usando o *benchmark* Linpack. Ao contrário de vários dos sistemas recentes na lista, o K não usa placas de processamento gráfico nem outros tipo de aceleradores. Por outro lado, os supercomputadores nas posições 2, 4 e 5 usam GPUs da NVIDIA nos cálculos. Além disso, 39 supercomputadores da lista também usam GPUs como aceleradores, este número era de apenas 17 na edição anterior. Destes 39 supercomputadores, 35 usam placas da NVIDIA, 2 usam processadores Cell e dois usam placas ATI Radeon. Observando que estes 39 supercomputadores são apenas 7,8% do total de 500, um outro dado relevante é que o poder de cálculo destes computadores corresponde a mais de 15% do poder total de toda a lista.

Recentemente, portanto, as atenções tem-se voltado ao estudo e à exploração do processamento paralelo fornecido por elementos de processamento gráfico (GPUs) para o processamento de aplicações de propósito geral, ou seja, aplicações científicas e não somente gráficas sendo executadas pela integração CPU-GPU. Esta integração, por sua vez, é realizada através da divisão do código a ser executado na CPU e na GPU. Para fins de aceleração de execução, porções menores do código, sejam elas sequenciais ou levemente paralelas podem ser escalonadas para a CPU através de modelos de programação como o OpenMP (CHAPMAN, 2007) e o código fortemente baseado em paralelismo de dados e laços que dominam o tempo de execução de uma aplicação, para uma GPU.

De maneira mais generalizada estamos tratando do que é chamada de Computação Heterogênea, que considera elementos de processamento, sejam CPUs *multicore* ou *manycore*, sejam GPUs *manycore* e por que não elementos de processamento reconfiguráveis (FPGAs), que estejam disponíveis na máquina

hospedeira (*host*) ou em um agrupamento de *hosts* (*cluster, grid*). Esses elementos especializados podem ser usados em conjunto com processadores tradicionais, funcionando como aceleradores para a execução de tarefas específicas.

A generalização de domínio é uma questão crítica, pois soluções ainda são bem específicas considerando-se o domínio da técnica nos elementos mais simples da hierarquia. A ideia de integrar esses elementos de processamento para a generalização de domínio possibilita soluções de problemas de complexidade maior, porém a tentativa de generalização do domínio não é uma tarefa trivial.

Máquinas paralelas exploram o paralelismo em diferentes níveis, realizando computação no nível de instruções, de dados ou de tarefas, o que leva a uma classificação conforme o nível no qual o hardware dá suporte ao paralelismo.

7.2.2. Classificações e terminologia

Esta seção abrange os principais termos e classificações usados para descrever tanto máquinas paralelas quanto os modelos de programação correspondentes. É importante para que o leitor se familiarize com os termos encontrados em qualquer texto da área. Alguns deles são apresentados a seguir.

É possível afirmar que a Taxonomia de Flynn (Flynn, 1972) é uma das mais clássicas da área de PAD, embora seja genérica e não abranja de forma detalhada todos os tipos de arquiteturas existentes hoje em dia. Entretanto, alguns de seus termos são usados até hoje para descrever o tipo de paralelismo encontrado em algumas arquiteturas:

- **SISD *(Single Instruction, Single Data)*:** Não expressam nenhum paralelismo, classe que representa as arquiteturas que trabalham com um único fluxo de instruções e um único fluxo de dado, remete ao Modelo de von Neumann com execução sequencial;
- **SIMD *(Single Instruction, Multiple Data)*:** Máquinas paralelas que executam exatamente o mesmo fluxo de instruções (mesmo programa) em cada uma das suas unidades de execução paralela, considerando fluxos de dados distintos;
- **MIMD *(Multiple Instruction, Multiple Data)*:** Máquinas paralelas que executam fluxos independentes e separados de instruções em suas unidades de processamento. Pode-se ter programas diferentes, independentes que processam entradas diferentes, múltiplos fluxos de dados.

Outra classificação essencial é aquela que divide as máquinas segundo o compartilhamento de memória. Esta classificação é importante, pois é diretamente responsável pelo modelo de programação a ser empregado no desenvolvimento das aplicações:

- **Multiprocessadores:** máquinas com memória compartilhada (UMA e NUMA, respectivamente com velocidades de acesso uniforme à memória e não uniforme), possibilitam o modelo de programação ***multithreaded***.
- **Multicomputadores**: máquinas onde cada elemento de processamento possui a sua própria memória (NORMA), exigem o modelo de programação através da **troca de mensagens** entre seus componentes.

Ainda existem outras classificações usuais considerando-se o nível de abstração:

- **SPMD *(Single Program, Multiple Data)*:** O modelo SPMD *(Single Program, Multiple Data)* foi definido por (Darema-Rogers and Pfister, 1985) (DAREMA, 2001), este modelo é mais geral que o modelo SIMD *(Single Instruction, Multiple Data)* e que o modelo *data-parallel*, pois o SPMD usa aplicações com paralelismo de dados e paralelismo de *threads*;
- **SIMT *(Single Instruction, Multiple Threads)*:** Modelo utilizado para gerenciar a execução de várias *threads* na arquitetura. Termo utilizado pela Nvidia para o CUDA;
- **MSIMD *(Multiple SIMD)*:** Máquinas que possuem um memória global de tamanho ilimitado, e *P* processadores SIMD independentes. Autores definiram (Bridges, 1990) e utilizam (Jurkiewicz and Danilewski, 2011) esta denominação.

7.3. Arquiteturas heterogêneas

Esta seção tem o objetivo de definir e apresentar as principais arquiteturas heterogêneas e contextualizar aquela que será foco no restante do capítulo, ou seja, a das GPUs.

Dongarra, 2009, propõe uma taxonomia para plataformas heterogêneas que pode ser utilizada para contextualizar as arquiteturas heterogêneas abordadas neste capítulo. Esta taxonomia inclui máquinas paralelas e sistemas distribuídos e possui as seguintes categorias em ordem crescente de heterogeneidade e complexidade:

- sistemas heterogêneos projetados por fabricantes específicos;
- clusters heterogêneos;
- redes locais de computadores;
- redes globais de computadores em um mesmo nível organizacional;
- redes globais de computadores de propósito geral.

Como é possível observar, existem várias classes de sistemas heterogêneos, onde a maioria recai em sistemas distribuídos (três últimas categorias). Este capítulo, entretanto, aborda apenas a primeira categoria, a de sistemas heterogêneos projetados por fabricantes específicos. O interesse, neste caso, são os dispositivos denominados *aceleradores* que atuam em conjunto com as CPUs dentro de um único nó de processamento.

Atualmente, observa-se a tendência em se utilizar uma quantidade maior de núcleos de processamento para se obter desempenho, ao invés de aumentar a frequência do *clock*. Em (Borkar, 2011) são apresentados o histórico dos últimos trinta anos e algumas perspectivas para as arquiteturas de microprocessadores para os próximos vinte anos. Num primeiro momento, os autores traçam um perfil dos últimos anos onde o desempenho era obtido a partir de um aumento na velocidade de *clock*, juntamente com a otimização das instruções e o aprimoramento da hierarquia de *cache*. Do ponto de vista das aplicações, os compiladores mantinham a tarefa de adaptar o código para executar num hardware mais sofisticado e obter melhores desempenhos.

Nos próximos vinte anos, porém, o principal fator que guiará os projetos de microprocessadores já é e continuará sendo o gerenciamento de energia. O problema é que, de uma maneira geral, quanto mais sofisticados os núcleos, maior o consumo de energia. Além disso, chegamos num momento em que esta sofisticação e o consequente maior gasto de energia já não traz mais tanto desempenho. Assim, a utilização de núcleos mais simples, que executam no modelo SIMD, juntamente com uma certa quantidade de núcleos mais sofisticados é a tendência natural para os próximos vinte anos. Esta abordagem tem sido caracterizada como computação heterogênea e é o foco deste capítulo.

No momento, os aceleradores são uma alternativa de baixo custo que já pode ser utilizada em conjunto com as CPUs de mercado para realizar computação heterogênea. Possuem a vantagem de melhorar o desempenho de uma série de aplicações altamente iterativas e que possuem um tipo paralelismo denominado "trivial", onde a dependência de dados não existe ou é mínima. Assim surge o modelo de computação heterogênea onde as tarefas são atribuídas ao hardware mais adequado presente na máquina. A seguir, são apresentados três exemplos de aceleradores. A arquitetura de GPU terá maior destaque na próxima seção.

7.3.1. Exemplos de aceleradores

Esta seção apresenta três exemplos de aceleradores, buscando salientar semelhanças e diferenças em suas arquiteturas, assim como os modelos de programação: Cell BEA, FPGAs e GPUs. As próximas seções apresentam com maior detalhe as GPUs da NVIDIA e o seu modelo de programação.

7.3.1.1 Cell BEA

O **Cell BEA** (*Cell Broadband Engine Architecture*) (Crawford, 2008) já apareceu como componente acelerador em máquinas no topo do TOP 500 (www.top500.org), o ranking que lista as máquinas com maior poder de processamento do mundo. Trata-se de uma arquitetura notória por equipar tanto o console do Playstation 3 quanto o *Roadrunner*, a primeira máquina a ultrapassar a marca dos Petaflops na lista do TOP 500. Sua configuração era composta por quatro CBE PowerXCell8i de 3.2 Ghz e dois *dual-core* Opteron de 1.8 Ghz em cada nó de processamento (Crawford, 2008).

O Cell BE é um processador heterogêneo equipado com uma CPU tradicional (PPE - *Power Processing Elemen*t) além de oito cores mais simples de propósito específico (SPE - *Synergistic Processing Elements*) em um mesmo chip.

A Figura 7.1 apresenta um esquema da arquitetura do Cell BE. Os principais elementos presentes são o PPE, os oito SPEs e os controladores de barramento e de memória. O barramento (EIB – *Element Interconect Bus*), em especial, é estruturado em dois anéis para cada direção que conectam todos os elementos.

O PPE é um núcleo de processamento típico, similar aos encontrados em máquinas baseadas em processadores Power. Em particular, as instruções de leitura e escrita na memória são realizadas através de uma típica hierarquia de cache. Os SPEs, entretanto, são núcleos menos complexos e menores, projetados para serem núcleos aceleradores que concentram a maior parte do poder de processamento do Cell BE (Kunzman, 2009). Ambos os tipos de núcleos possuem instruções do tipo SIMD em seu

conjunto de instruções, o que ajuda a acelerar as aplicações que possuam características vetoriais.

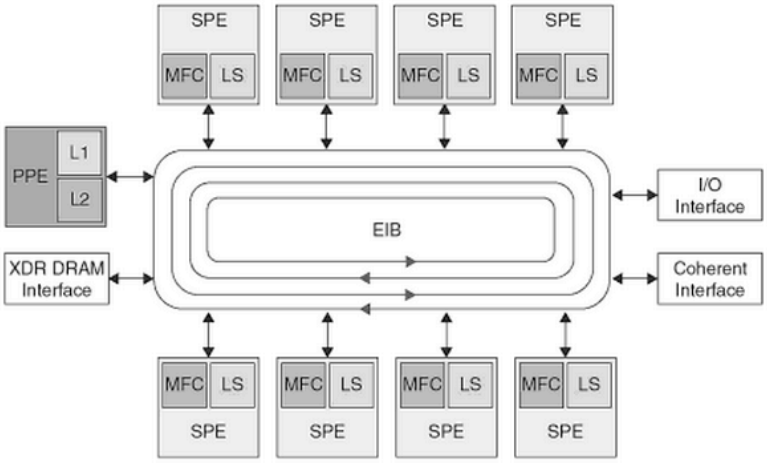

Figura 7.1: componentes do Cell BEA (Fonte: Dongarra, 2009)

A programação dos SPEs, entretanto, é considerada um tanto complexa. Em primeiro lugar, os SPEs não possuem hierarquia de cache. Ao invés disso, possuem uma memória local (*local storage*) que deve ser gerenciada explicitamente no código da aplicação. Em segundo lugar, as transferências de memória devem ser realizadas via DMA (*Direct Memory Access*) que também devem ser iniciadas de forma explícita no código da aplicação. Por último, os SPEs tem seu próprio conjunto de instruções (ISA – *Instruction Set Architecture*), o que faz com que seu código tenha que ser compilado separadamente do código principal da aplicação que executa no PPE e é responsável por iniciar as execuções nos SPEs (Kunzman, 2009).

Embora seja um bom exemplo de arquitetura heterogênea utilizada para PAD, o processador que equipou o *Roadrunner* (o PowerXCell8i) já não será utilizado pela IBM para construir supercomputadores (Wolfe, 2012), fato que torna esta arquitetura uma escolha um tanto duvidosa para uso em PAD.

7.3.1.2 FPGA

FPGAs (*Field Programmable Gate Array Architecture*) tem sido empregados recentemente como aceleradores para PAD devido aos avanços obtidos principalmente nas tecnologias de interconexões de alta velocidade (Brodtkorb, 2010). Embora sua arquitetura seja mais simplificada do que a dos demais aceleradores, permite um alto grau de paralelismo de dados em instruções como adições e multiplicações.

No contexto de PAD, alguns dos maiores fabricantes de FPGAs, tais como a Xilink e a Altera, tem colaborado com fabricantes como a Convey e a Cray que oferecem placas compatíveis com processadores Intel ou AMD usando o mesmo barramento de alta velocidade e memória na placa.

Um FPGA consiste num conjunto de blocos lógicos reconfiguráveis, além de blocos de processamento digital de sinais e opcionalmente um ou mais núcleos de CPU tradicionais todos conectados por uma interconexão extremamente flexível. Esta interconexão é reconfigurável e pode ser utilizada para adaptar as aplicações às FPGAs.

Quando configuradas, as FPGAs funcionam com os circuitos integrados feitos para aplicações específicas (ASICs – *application specific integrated circuits*) (Brodtkorb, 2010). A Figura 7.2 apresenta um esquema genérico para um sistema heterogêneo composto de uma CPU e de um FPGA. Existem versões compatíveis com CPUs Intel e AMD.

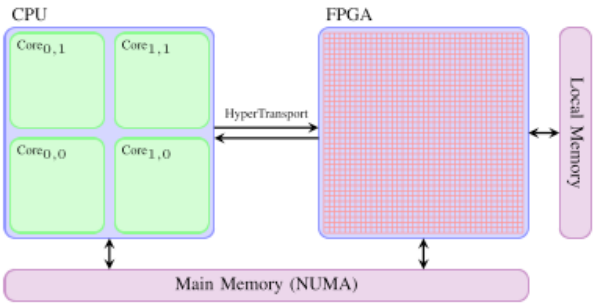

Figura 7.2: sistema heterogêneo com CPU e FPGA (Fonte: Brodtkorb, 2010)

Do ponto de vista do uso de FPGAs para PAD, nem todas as aplicações podem facilmente obter *speedup*. Embora (Storaasli, 2007) descreva ganhos de até 100x obtidos em uma aplicação de biologia computacional (comparação de sequências de DNA), o modelo de programação ainda é considerado complexo.

As principais ferramentas de programação são as linguagens VHDL e Verilog, que exigem um conhecimento especializado. Além destas, (Brodtkorb, 2010) ainda cita, entre outras, Mitrion-C e System-C, que permitem programação em linguagem C com extensões para acesso à FPGA. Outra opção é a ferramenta Viva, que possui um ambiente gráfico de programação similar ao do Labview.

7.3.1.3 GPU

O exemplo mais contundente de acelerador no momento são as **GPUs** (*Graphics Processing Units*). Inicialmente possuíam apenas função gráfica, mas rapidamente seu potencial foi percebido e versões para processamento genérico (GPGPU – *General Purpose GPU*) surgiram. Hoje em dia é possível observar algumas máquinas equipadas com centenas de GPUs no topo da lista do TOP 500.

As GPUs são dispositivos aceleradores que atuam em conjunto com as CPUs. Os dois principais fabricantes de GPUs no momento são a NVIDIA e a AMD que podem atuar em conjunto com CPUs Intel ou AMD. O paralelismo é do tipo SIMD, onde milhares de *threads* executam simultaneamente nas centenas de núcleos presentes na GPU. O programa principal executa na CPU (*host*) e é o responsável por iniciar as *threads* na GPU (*device*). As GPUs tem sua própria hierarquia de memória e os dados devem ser transferidos através de um barramento *PCI express*. A Figura 7.3 mostra um esquema genérico de um sistema heterogêneo composto de uma CPU e de uma GPU.

A NVIDIA é a principal fabricante de GPUs para PAD e equipa algumas máquinas presentes entre as dez mais rápidas do mundo, por exemplo. Além de serem aceleradores eficientes, as GPUs tem um baixo consumo de energia comparados a *multicores*, por isso também são opções adotadas em "computação verde" (*green computing*) (*www.green500.org*).

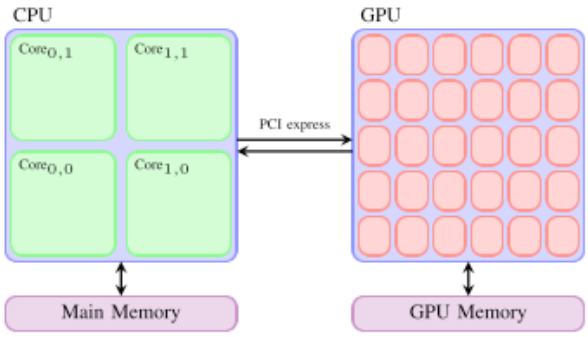

Figura 7.3: GPU em combinação com uma CPU (Fonte: Brodtkorb, 2010)

Os principais ambientes de programação para GPUs são CUDA (NVIDIA) e OpenCL (padrão aberto para computação heterogênea). Enquanto CUDA pode ser utilizada apenas em placas da NVIDIA, OpenCL é um padrão que pode ser utilizado com uma série de dispositivos, o que inclui as placas da NVIDIA e da AMD. OpenCL é um ambiente um pouco mais recente e somente as últimas versões começam a ter desempenho suficiente para alcançar os obtidos por CUDA. Um dos principais problemas de OpenCL é manter o desempenho enquanto preserva a portabilidade entre dispositivos diferentes (Du, 2010).

As próximas seções irão detalhar a arquitetura e as opções de ferramentas de desenvolvimento para GPUs, por serem as opções mais populares em termos de aceleradores hoje em dia. Em termos de arquitetura, a ênfase será dada à arquitetura Fermi da NVIDIA, enquanto que os ambientes de programação abordados para GPU serão CUDA, OpenCL e OpenACC.

7.4. Arquitetura das GPUs

Esta seção apresenta com maior detalhe a arquitetura das GPUs, em especial a Fermi da NVIDIA, que será usada como exemplo principal. Esta arquitetura permite a execução concorrente de *kernels*, entre outras novidades com relação às suas antecessoras. A Figura 7.4 demonstra o crescimento comparativo da capacidade de realização de operações de ponto flutuante por segundo (FLOPS/s) entre GPU e CPU. A série GTX 400 introduziu o uso da arquitetura Fermi, enquanto que a série GTX 600 introduziu a arquitetura Kepler, que serão vistas neste capítulo.

As GPUs são compostas de centenas de núcleos (*cores*) simples que executam o mesmo código através de centenas a milhares de *threads* concorrentes. Esta abordagem se opõe ao modelo tradicional de processadores *multicore*, onde algumas unidades de núcleos completos e independentes são capazes de processar *threads* ou processos. Estes núcleos completos, as CPUs, possuem poderosas unidades de controle e de execução capazes de executar instruções paralelas e fora de ordem, além de contarem com uma poderosa hierarquia de cache. Já as GPUs contam com unidades de controle e de execução mais simples, onde uma unidade de despacho envia apenas uma instrução para um conjunto de núcleos que a executarão em ordem. O modelo de execução das GPUs é conhecido como SIMT (*Single Instruction Multiple Threads*), derivado do clássico termo SIMD (*Single Instruction Multiple Data*).

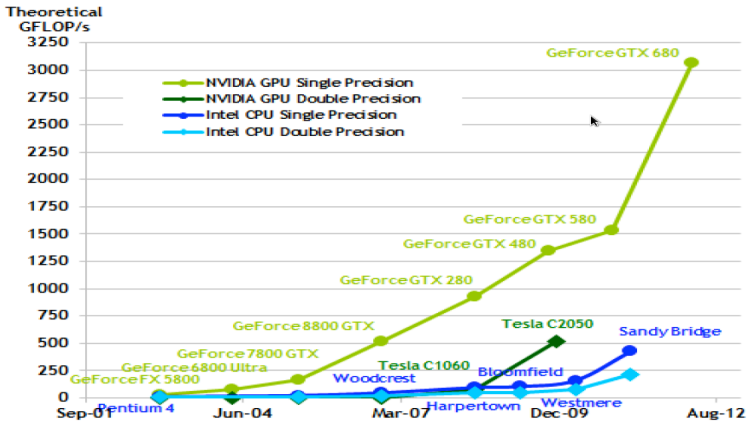

Figura 7.4: evolução de desempenho das GPUs (fonte: NVIDIA, 2012a)

Este modelo exige uma nova postura por parte dos desenvolvedores de software, portando é importante que esta arquitetura seja desvendada antes de apresentar o modelo de programação.

Outra característica importante é a hierarquia de memória. As GPUs, possuem memória global que pode ser acessada por todas as *threads*, porém as mais modernas já contam com caches de nível 1 e de nível 2. Além disso, outras memórias especializadas também podem ser usadas para acelerar o processamento.

7.4.1. Arquitetura Fermi

A arquitetura Fermi da NVIDIA segue este princípio de dedicar uma maior quantidade de transístores às unidades de execução, ao invés de dedica-los às unidades de controle e cache. A Figura 7.5 apresenta uma visão geral da arquitetura Fermi.

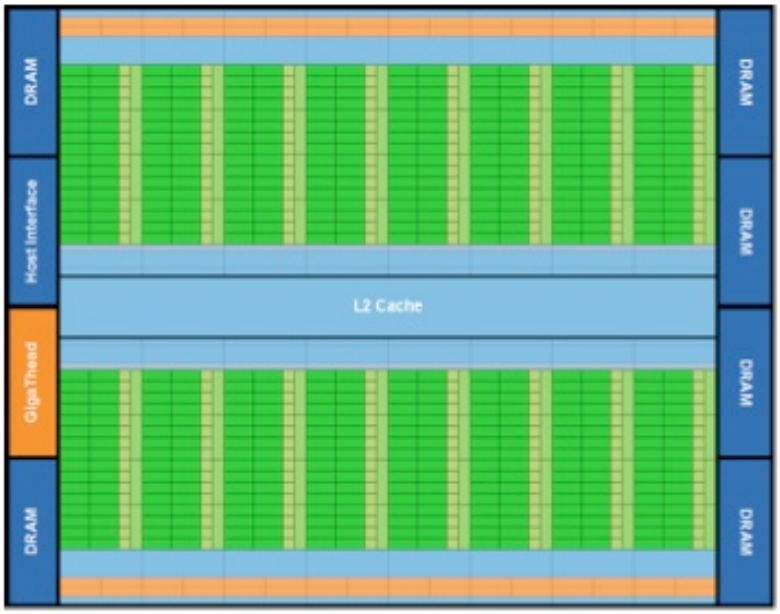

Figura 7.5: visão geral da arquitetura Fermi (fonte: NVIDIA, 2009)

Esta arquitetura (Figura 7.5) conta com 16 SM (*Streaming Multiprocessors*), cada

um composto por 32 núcleos de processamento (*cores*), resultando num total de 512 núcleos. É possível observar uma cache de segundo nível (L2) compartilhada por todos os SM. A cache de primeiro nível (L1) é compartilhada pelos 32 núcleos de cada SM. A memória compartilhada (*shared memory*) pode ser usada explicitamente pelo programador como uma memória de "rascunho" que pode acelerar o processamento de uma aplicação, dependendo do seu padrão de acesso aos dados. Esta memória é dividida fisicamente com a cache de primeiro nível com um total de 64KB, cujo tamanho é configurável: 16 KB – 48KB para cache e memória compartilhada respectivamente ou ao contrário. Além dos dois níveis de cache e da memória compartilhada, a Fermi conta com uma memória global (DRAM) de até 6GB.

A Figura 7.6 apresenta os principais componentes de um SM que compõe a arquitetura Fermi, apresentando seus núcleos simplificados, as unidades de escalonamento e despacho de instruções, os registradores, a cache de nível 1, entre outros.

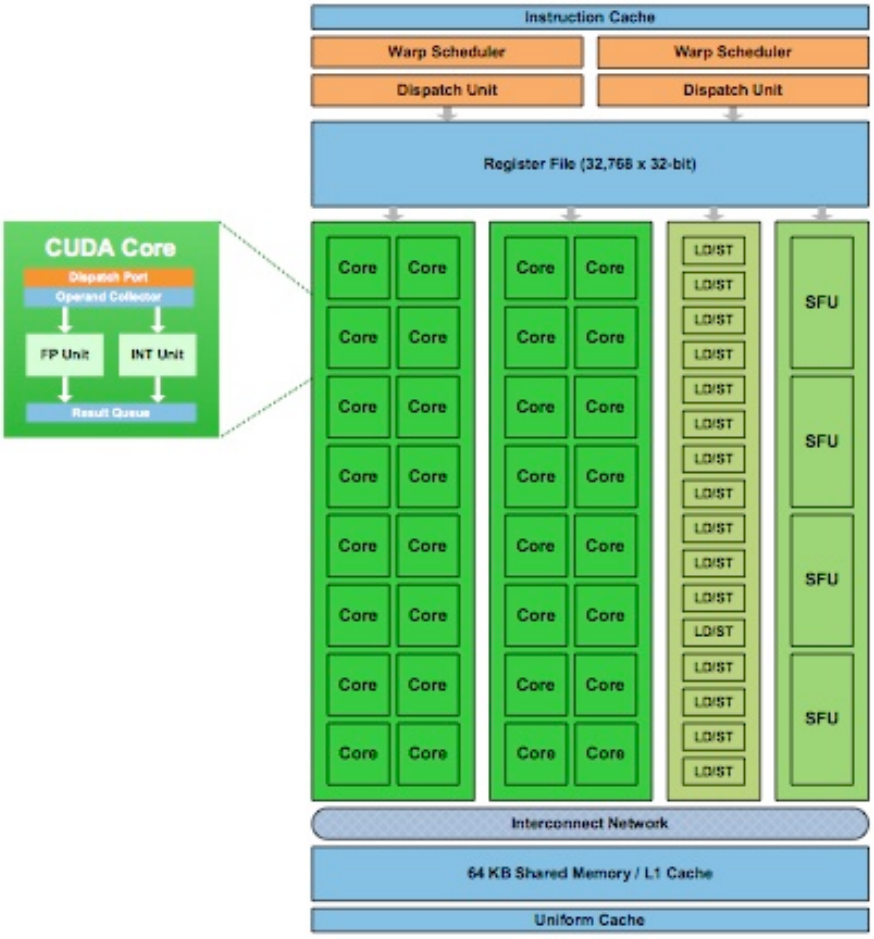

Figura 7.6: arquitetura de um SM (fonte: NVIDIA, 2009)

Cada SM é composto por quatro blocos de execução controlados por duas unidades escalonamento de *warps* (grupos de 32 *threads*): dois blocos de 16 núcleos simples, capazes de executar em ordem uma instrução inteira ou de ponto flutuante, um bloco de 16 unidades *load/store* e um bloco de 4 unidades de processamento de instruções

especiais (SFU – *Special Function Units*). Além disso, cada SM conta com uma memória cache L1/memória compartilhada de 64KB, já mencionada, e com 32KB de registradores, compartilhados entre todas as *threads* que executarão no SM.

Uma GPU, portanto, é composta basicamente de um conjunto de multiprocessadores (SMs) conectados a uma cache de nível 2 e a uma memória global. Detalhes desta arquitetura, assim como características que permitem a execução concorrente de diferentes unidades de código (*kernels*), podem ser verificados pelo desenvolvedor com funções disponíveis pela API do CUDA *Runtime* e pela API do CUDA *Driver*.

O escalonamento de *threads* é feito em grupos de 32 *threads* que executam em paralelo (*warps*). Na arquitetura Fermi cada multiprocessador tem dois escalonadores de *warps* e duas unidades de despacho de instruções, possibilitando que dois *warps* possam ser despachados e executados concorrentemente (*dual warps*) (NVIDIA Corporation, 2009).

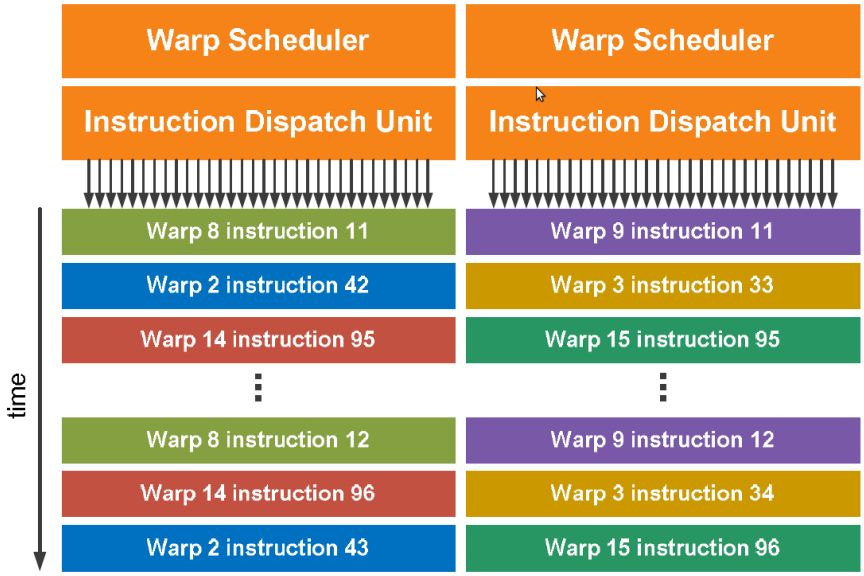

Figura 7.7: Fermi *Dual Warp* (NVIDIA Corporation, 2009)

O escalonador seleciona dois *warps*, como estes são independentes é possível despachar uma instrução de cada *warp* para o grupo de 16 cores, sem a preocupação de dependências entre as instruções, o que melhora o desempenho. A Figura 7.7 apresenta o esquema de escalonamento da arquitetura Fermi.

7.4.2. Arquitetura Kepler

A Arquitetura Kepler (NVIDIA Corporation, 2012b) é a nova arquitetura implementada nas GPUs da NVIDIA. Para a Kepler GK110 as características mais importantes estão relacionadas às melhorias nos multiprocessadores, paralelismo dinâmico e a tecnologia *Hyper Q*.

As GPUs com implementações mais completas da arquitetura trazem um conjunto de até 15 multiprocessadores (SMX) com 192 núcleos cada. A organização destes núcleos em um multiprocessador pode ser vista na Figura 7.8.

Figura 7.8: Multiprocessador (SMX) Kepler (Fonte: NVIDIA, 2012b).

Essa nova organização de mais autonomia à GPU, como o conceito de Paralelismo Dinâmico introduzido pela arquitetura Kepler, a GPU pode dinamicamente disparar a execução de novas *threads*, sincronizar resultados e controlar o escalonamento, adaptando-se ao fluxo de execução sem a necessidade de envolver o programa executado na CPU do *host*. Esta independência, possibilita que mais de um programa possa ser executado diretamente na GPU, e ainda que *kernels* possam fazer chamadas a outros *kernels* criando os recursos necessários para a execução deles, independente do código executado no *host*. Esta ideia de paralelismo dinâmico pode ser vista na Figura 7.9, onde são comparados os modelos de execução de *threads* nas arquiteturas Fermi e Kepler.

O comparativo do modelo de execução apresentado na Figura 7.9, mostra a autonomia dada às GPUs da arquitetura Kepler. Enquanto que nas arquiteturas anteriores existia uma dependência do código principal executado no *host*, para

sincronizar dados e disparar novos *kernels*, na arquitetura Kepler uma chamada à uma função *kernel* pode gerar novas chamadas a outros *kernels* feitas pela própria GPU, que se adapta ao fluxo de execução.

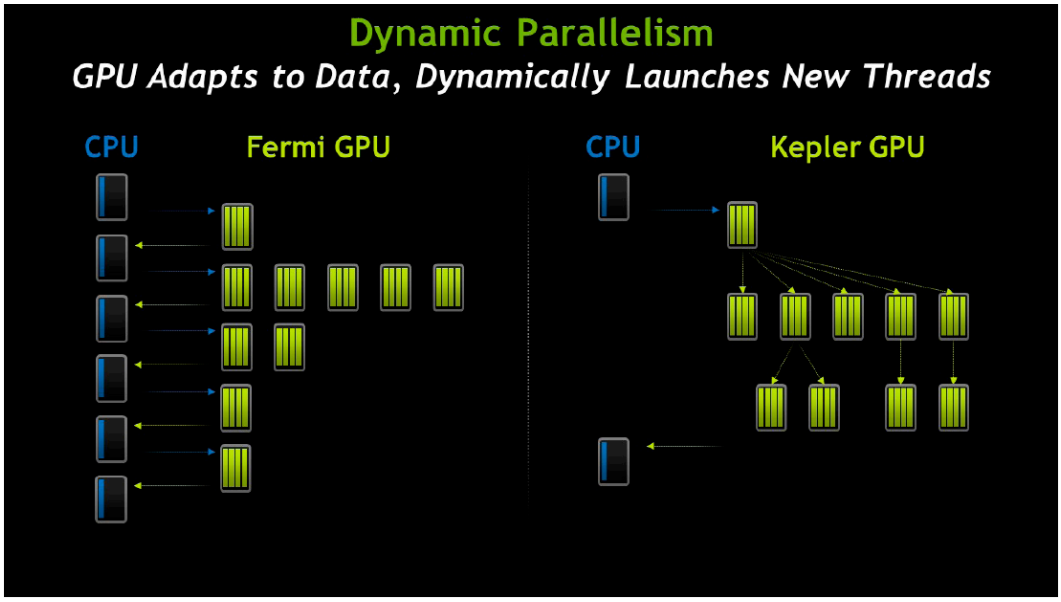

Figura 7.9: Paralelismo Dinâmico (Fonte: NVIDIA Corporation, 2012b)

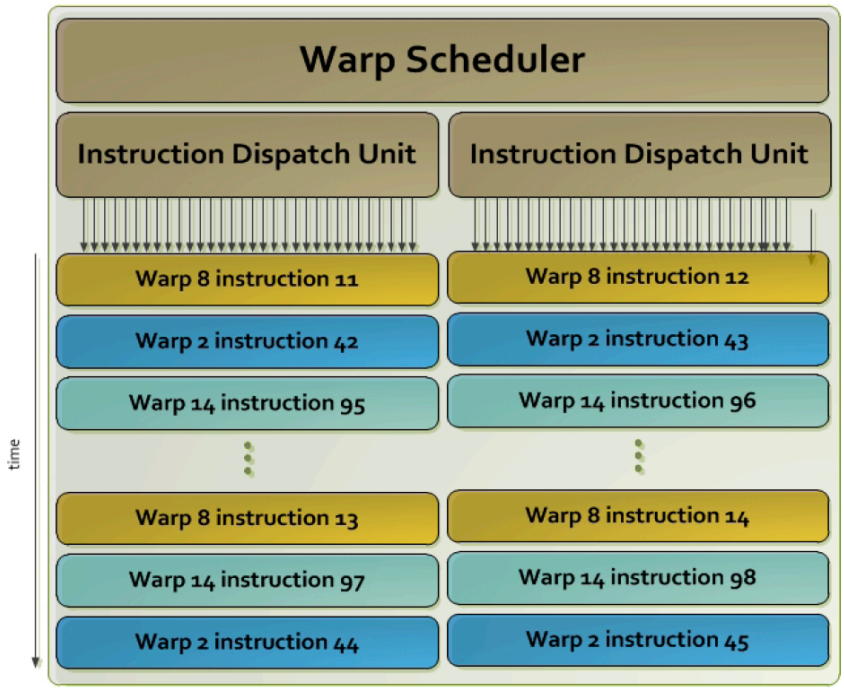

Figura 7.10: Kepler Quad-Warp (Fonte: NVIDIA Corporation, 2012b)

A tecnologia Hyper-Q possibilita que diferentes *threads* do *host* possam disparar a execução de *kernels* simultaneamente, na Kepler GK110 são possíveis 32 conexões simultâneas com os cores da CPU, contra 1 conexão das arquiteturas anteriores (NVIDIA Corporation, 2012b). A proposta é que a GPU possa ser utilizada na aceleração de programas escritos para plataformas de troca de mensagens, com MPI, cujos processos poderão disparar a execução simultânea de *kernels* na GPU, melhorando o desempenho dessas aplicações.

Quanto ao escalonamento, na arquitetura Kepler cada SMX possui quatro escalonadores de *warps* e oito unidades de despacho de instruções, possibilitando que quatro *warps* (4 x 32 *threads* paralelas) possam ser escalonados e executados concorrentemente (*quad warp scheduler*). A organização de cada um dos escalonadores é mostrado na Figura 7.10, cada um deles com duas unidades de despacho de instruções.

7.4.3. Detecção do dispositivo instalado

Para verificar as características dos dispositivos instalados em um *host*, a API do CUDA fornece funções que recuperam as propriedades da GPU. A Figura 7.11 apresenta a saída da execução do exemplo deviceQuery disponível com o kit de desenvolvimento CUDA, o código procura por dispositivos instalados no host que tenham suporte a CUDA, no caso GPUs e então lista suas propriedades.

```
$ ./deviceQuery
[deviceQuery] starting...

./deviceQuery Starting...
 CUDA Device Query (Runtime API) version (CUDART static linking)

Found 1 CUDA Capable device(s)

Device 0: "GeForce GTX 260"
  CUDA Driver Version / Runtime Version          4.2 / 4.2
  CUDA Capability Major/Minor version number:    1.3
  Total amount of global memory:                 895 MBytes (938803200 bytes)
  (27) Multiprocessors x (  8) CUDA Cores/MP:    216 CUDA Cores
  GPU Clock rate:                                1242 MHz (1.24 GHz)
  Memory Clock rate:                             999 Mhz
  Memory Bus Width:                              448-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768),
3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers              1D=(8192) x 512,
2D=(8192,8192) x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
```

```
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Concurrent kernel execution:                   No
  Alignment requirement for Surfaces:            Yes
  Device has ECC support enabled:                No
  Device is using TCC driver mode:               No
  Device supports Unified Addressing (UVA):      No
  Device PCI Bus ID / PCI location ID:           1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >


deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.2, CUDA Runtime
Version = 4.2, NumDevs = 1, Device = GeForce GTX 260
[deviceQuery] test results...
PASSED


> exiting in 3 seconds: 3...2...1...done!
```

Figura 7.11: Saída da execução do *deviceQuery* (Fonte: CUDA SDK, 2012)

Pelas propriedades listadas na Figura 7.11 podemos verificar as características e capacidades da GPU disponível. Características como a quantidade de memória, a quantidade de multiprocessadores, a quantidade de *cores* por multiprocessador, a quantidade de *threads* paralelas em execução (*warp size*), quantidades máximas de *threads* por multiprocessador, por bloco, dimensões do grid, dimensões do bloco e se a GPU suporta execução concorrente de *kernels*.

Outra característica importante é a capacidade de computação (*compute capability*), que basicamente faz um mapeamento entre o hardware da placa e a plataforma de desenvolvimento CUDA. No exemplo da Figura 7.11 a capacidade computacional é 1.3. O manual de desenvolvimento CUDA (NVIDIA, 2012b) indica quais são as características disponíveis para esta capacidade de computação, assim como apresenta uma tabela comparativa entre as capacidades.

7.5. OpenMP: *threads* na CPU

O objetivo desta seção é descrever as principais diretivas do conjunto OpenMP (*Open Multi-Processing*) (Chapman, 2008), ressaltando o uso de *threads* em arquiteturas *multicore*. Antes de descrever o modelo de programação em aceleradores do tipo GPU é importante descrever o modelo de programação paralelo das CPUs tradicionais, salientando que estes podem ser combinados.

O processamento paralelo na máquina hospedeira utilizando CPU já é bem antigo, indo das *threads* do padrão POSIX até bibliotecas que exploram técnicas de memória compartilhada. A interface ***pthreads*** (POSIX - *Portable Operating System Interface for UNIX*), por exemplo, oferece suporte para a criação e sincronização de *threads*.

O OpenMP possui vantagens como a facilidade de programação e tem se tornado a alternativa mais popular para programação paralela de alto desempenho em SMPs. Por este motivo foi escolhida para ilustrar a programação *multicore* neste texto.

O padrão é composto por um pequeno conjunto de diretivas de programação mais um pequeno conjunto de funções de biblioteca e variáveis de ambiente que usam como base as linguagens C/C++ e Fortran. Trata-se de um padrão, portanto várias implementações estão disponíveis. É comum que compiladores já conhecidos, como o próprio **gcc**, possuam opções de compilação para OpenMP.

7.5.1. Modelo de execução

Um processo em execução é composto por segmentos de código, dados, pilha e *heap* (área de memória livre), no mínimo. Cada processo contém uma *thread*, que é composta basicamente por um contexto (estado do hardware, informações ao SO), o código e uma pilha. Se o processo tiver mais de uma thread, todas elas compartilham os segmentos de código, dados e *heap*. Embora o código seja compartilhado, cada *thread* pode executar uma linha de código diferente, pois cada uma tem seu próprio contexto. O segmento de pilha também não é compartilhado, o que permite que as *threads* mantenham seus dados locais.

Uma *thread* normalmente é criada a partir do endereço de um procedimento que é passado como ponto de partida para sua execução (exemplo: *pthreads*). Em OpenMP, porém, as *threads* são criadas a partir de fragmentos de código (blocos) anotados que normalmente correspondem a alguma tarefa iterativa. O OpenMP se encarrega de criar o procedimento executado pelas *threads*. Este procedimento também é chamado de *closure*, pois tem acesso a variáveis declaradas no escopo externo.

No início do processamento há uma única *thread* (*master thread*) que executa sequencialmente até encontrar a primeira construção que delimita uma região paralela (*parallel region*). Uma operação similar a um *fork* acontece então, fazendo com que um **time de *threads*** seja criado para executar o bloco que representa a região paralela (a thread mestre participa do time). Quando o time de *threads* termina de executar todos os comandos do bloco acontece uma operação semelhante ao *join* que sincroniza e finaliza todas as *threads* criadas e permite que apenas a thread mestre avance. A Figura 7.12 demonstra este comportamento através de duas regiões paralelas (um programa OpenMP pode ter quantas regiões paralelas quantas forem necessárias).

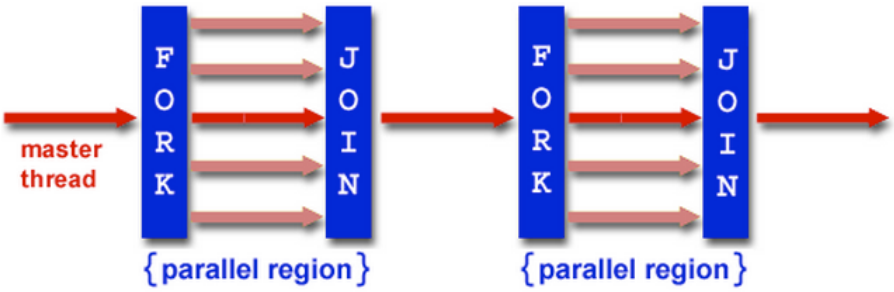

Figura 7.12: Modelo *fork-join* usado pelo OpenMP (Fonte: Barney, 2012)

7.5.2. Modelo para as diretivas em C/C++

As diretivas em C/C++ para OpenMP estão contidas em diretivas do tipo **#pragma**, que são dirigidas ao pré-processador, mais a palavra chave **omp** juntamente com o nome da diretiva OpenMP. O formato genérico para uma diretiva OpenMP em C/C++ pode ser visto na Tabela 7.1.

Tabela 7.1: Formato de uma diretiva OpenMP (Fonte: Barney, 2012)

#pragma omp	Diretiva	Atributos (cláusulas)	Nova linha
Obrigatório para todas as diretivas OpenMP C/C++.	Diretiva OpenMP válida.	Campo opcional. Os atributos podem aparecer em qualquer ordem.	Obrigatório. Precede uma estrutura de bloco.

A principal diretiva do OpenMP é a **parallel**. Esta diretiva identifica uma região de código (bloco) que será executada por múltiplas threads. A Figura 7.13 mostra um exemplo de programa em OpenMP onde se pode observar o uso da diretiva **parallel** combinada com a diretiva **for**, que paraleliza automaticamente o laço que vem em seguida. Algumas cláusulas também foram utilizadas no exemplo e serão explicadas a seguir. O exemplo utiliza valores reduzidos para que a saída possa ser visualizada e analisada com maior facilidade. Na prática, a carga de trabalho das *threads* deve ser maior para que o paralelismo seja melhor explorado.

```
01 #include <omp.h> //include necessário em programas OpenMP
02 #include <stdio.h>
03 #define SIZE 16
04
05 int main() {
06    int A[SIZE], i;
07
08    //inicia as threads que executarão o bloco "for"
09    #pragma omp parallel for schedule(static, 2) num_threads(4)
10             for(i = 0; i < SIZE; i++){
11                   A[i] = i * i;
12                   printf("Th%d[%d] = %d\n", omp_get_thread_num(),
13                            i, A[i]);
14             }
15 }
```

Figura 7.13: Exemplo de programa OpenMP: criação de um vetor de quadrados.

Na Figura 7.13 podem ser identificadas algumas das principais diretivas e funções de OpenMP. São elas:

- inclusão do cabeçalho OpenMP (linha 01);
- uso da principal diretiva OpenMP, a **parallel** (linha 9), que automaticamente cria as *threads* que executarão o bloco de instruções (o laço **for**, no caso);
    - a **cláusula num_threads** está associada à diretiva **parallel** e serve para definir a quantidade de *threads* que será lançada;
    - caso ela não seja usada, a quantidade de *threads* será a definida pela variável de ambiente **OMP_NUM_THREADS** (normalmente ela é associada à quantidade de núcleos da máquina);
    - outra alternativa é utilizar a função de biblioteca **omp_set_num_threads()** antes da primitiva **parallel**.
- uso da diretiva **for** (linha 9) automaticamente divide a execução do laço descrito logo em seguida no código (linha 10);
    - a **cláusula schedule** indica o tipo de escalonamento que será realizado entre as *threads*: os índices podem ser atribuídos de forma estática (***static***) ou dinâmica (***dynamic***). A tabela 2 mostra dois exemplos de execução com o tamanho da tarefa (*chunk*) igual a 2.
- o uso da função **omp_get_thread_num()**(linha 12) retorna o identificador da *thread* chamadora entre o time de *threads* que foram iniciadas pela diretiva **parallel** – todas são numeradas de `0` a `num_threads -1`.

Tabela 7.2: efeito das opções *static* e *dynamic* no exemplo da Figura 7.13.

Exemplo de execução: *static*	Exemplo de execução: *dynamic*
Th0[0] = 0	**Th1[0] = 0**
Th0[1] = 1	Th0[2] = 4
Th0[8] = 64	Th2[4] = 16
Th0[9] = 81	Th3[6] = 36
Th1[2] = 4	**Th1[1] = 1**
Th1[3] = 9	Th0[3] = 9
Th1[10] = 100	Th2[5] = 25
Th1[11] = 121	Th3[7] = 49
Th2[4] = 16	Th1[8] = 64
Th2[5] = 25	Th0[10] = 100
Th2[12] = 144	Th2[12] = 144
Th2[13] = 169	Th3[14] = 196
Th3[6] = 36	Th1[9] = 81
Th3[7] = 49	Th0[11] = 121
Th3[14] = 196	Th2[13] = 169
Th3[15] = 225	Th3[15] = 225

A Tabela 7.2 apresenta duas saídas para a execução do exemplo da Figura 7.13. Em primeiro lugar, é necessário observar que as *threads* concorrem por uma série de recursos da máquina, entre eles o console de saída. Assim, não é possível afirmar que a ordem em que as impressões aparecem na tela é a mesma ordem de execução, mesmo porque numa máquina *multicore* temos paralelismo real e as instruções podem ser executadas exatamente ao mesmo tempo. Porém, o console é um só e teremos concorrência pelo seu uso, o que dá um efeito "bagunçado" à saída do programa. Além disso, a cada execução podemos ter impressões (e execuções) em ordens diferentes.

Na primeira coluna da Tabela 7.2 temos a saída de uma execução do **parallel for** com a opção **static** para a cláusula **schedule**. Neste exemplo, o *chunk* (segundo parâmetro da cláusula **schedule**) foi definido com o valor 2. Isto significa que os índices serão distribuídos em "pacotes" de dois elementos para cada *thread*. O *chunk* pode ser definido como sendo o tamanho da tarefa de cada *thread*. Nesta coluna, podemos observar o tipo de escalonamento *round robin* empregado na opção **static**: o escalonamento é feito atribuindo-se um pacote de dois elementos para cada *thread* de forma circular na ordem dos identificadores das *threads*. A primeira rodada de entrega está destacada em negrito na primeira coluna. A thread 0 (Th0) recebe o primeiro pacote com os índices 0 e 1, a thread 1 (Th1) recebe o segundo pacote com os índices 2 e 3, e assim por diante.

Se o tamanho do *chunk* não for especificado, ele será calculado pelo sistema de forma a dividir o total de itens (quantidade de iterações do laço *for*) pela quantidade de *threads* no time. Assim, cada *thread* receberá no máximo um pacote de tarefas (é importante para o balanceamento que a quantidade de iterações seja divisível pelo número de *threads* lançadas).

Na segunda coluna da Tabela 7.2 temos o exemplo para a opção **dynamic** da cláusula **schedule**. Neste caso, os *chunks* de dois elementos foram distribuídos durante a execução às *threads* prontas para executá-los. O primeiro pacote, no exemplo, foi entregue à *thread* 1 (Th1) ao invés da *thread* 0 (Th0) como no exemplo anterior (este comportamento foi destacado em negrito na segunda coluna). Nesta opção de escalonamento, caso o tamanho do *chunk* não seja definido, ele será considerado como sendo de tamanho 1.

A compilação é simples. No gcc, por exemplo, basta acrescentar a opção **fopenmp** ao comando de compilação. A Figura 7.14 mostra um possível comando de compilação no **gcc**.

```
gcc –fopenmp quadrados.c –o quadrados
```

Figura 7.14: Comando de compilação usando gcc

Ferramentas de programação como o Microsoft Visual Studio 2010 ou o Apple Xcode, por exemplo, possuem opção para ativação do suporte ao OpenMP. Normalmente isto pode ser definido editando-se as propriedades do projeto.

7.5.3. Modelo de memória

No exemplo da Figura 7.13 foram criadas quatro *threads* e todas elas acessam o vetor **A**. Isto é possível porque as variáveis declaradas no escopo onde o bloco paralelo se encontra são automaticamente compartilhadas entre as *threads*. Porém, se observarmos atentamente o exemplo, veremos que isso não poderia ser aplicado à variável **i** que é usada como índice no laço **for**. Caso esta variável fosse compartilhada, teríamos várias *threads* tentando escrever em posições sobrepostas no vetor. Isto não acontece no exemplo porque usamos a diretiva **for**, que transforma automaticamente o índice do vetor em variável local que não é compartilhada entre as *threads*.

É possível especificar de forma explícita o comportamento das variáveis que serão usadas dentro do bloco **parallel** através das cláusulas **shared** e **private**. Embora não seja necessário no exemplo, a Figura 7.15 mostra como ficaria a utilização destas cláusulas no código da Figura 7.13 (linha 9).

```
#pragma omp parallel for schedule(static, 2) num_threads(4) shared(A)
private(i)
```

Figura 7.15: Exemplo de uso das cláusulas shared **e** private

Um problema quando se programa com memória compartilhada é controlar o uso das variáveis compartilhadas que sofrem atualização a partir de várias *threads*. Vamos

supor que ao invés de criar um vetor de quadrados queremos realizar o somatório de quadrados. O trecho da Figura 7.16 foi executado algumas vezes com SIZE igual a 1000 elementos e durante uma pequena série de execuções foi possível observar resultados diferentes (errôneos) para o somatório, resultado da falta de proteção para a variável **somatorio**.

Para realizar este tipo de teste é possível desativar a opção para suporte ao OpenMP a fim de se obter o resultado sequencial, livre de possíveis *bugs*. Para este exemplo, resultado do somatório dos primeiros 1000 quadrados é 332833500 na versão sequencial. Nas quatro primeiras execuções paralelas com o OpenMP ativado o resultado foi o mesmo. Porém, já na quinta execução obteve-se um resultado diferente: 316693566, ou seja, um resultado errado. O programa, desta forma, é considerado não-determinístico: muitas vezes o *bug* só é detectado em situações muito específicas, tornando o problema difícil de detectar e corrigir.

```
int somatorio=0, i;
#pragma omp parallel for schedule(static,2) num_threads(4) //
 shared(somatorio)
      for(i = 0; i < SIZE; i++)
            somatorio += i * i;
printf("somatorio = %d\n", somatorio);
```

Figura 7.16: Variável compartilhada sem proteção

O OpenMP oferece pelo menos duas maneiras de prevenir esta situação. A primeira é a primitiva **critical**, que cria uma área de exclusão mútua que delimita um bloco de código onde apenas uma *thread* poderá executar de cada vez. Esta primitiva seria correspondente a se colocar um *lock* no início do bloco e um *unlock* no final. A Figura 7.17 ilustra o uso da primitiva **critical**.

```
int soma_local, somatorio=0, i;

#pragma omp parallel num_threads(4) shared(somatorio) //
private(soma_local)
{
soma_local=0;
      #pragma omp for schedule(static, 2)
            for(i = 0; i < SIZE; i++)
                  soma_local += i * i;
      #pragma omp critical
            somatorio += soma_local;
}
```

Figura 7.17: Exemplo da primitiva critical

Para agilizar o processamento, uma variável local foi criada. Ela permite que as *threads* realizem o processamento local de forma independente uma das outras. Ao final deste processamento local, a variável compartilhada (**somatorio**) é atualizada por uma *thread* de cada vez. Este tipo de processamento é conhecido como *operação de redução* (*reduction*).

Quando se tratar de uma diretiva **for**, entretanto, visto que esta é uma operação comum, é possível utilizar a cláusula **reduction** que realiza a operação de redução de

forma implícita. O resultado será compartilhado e não é necessário especificar através da cláusula *shared*. A Figura 7.18 mostra uma nova versão do código, agora com o uso da cláusula **reduction**.

```
#pragma omp parallel for schedule(static, 2) reduction(+:somatorio)
            for(i = 0; i < SIZE; i++)
                  somatorio += i * i;
```

Figura 7.18: Exemplo de uso da cláusula *reduction*

O OpenMP possui ainda uma série de primitivas e cláusulas que podem ser utilizadas para gerenciar a execução do código paralelo. Uma descrição completa pode ser encontrada em (Chapman, 2008).

7.6. CUDA: *threads* na GPU

CUDA (*Compute Unified Device Architecture*), é a tecnologia da NVIDIA introduzida em Novembro de 2006 conhecida como Arquitetura de Computação Paralela de Propósito Geral. CUDA trouxe um novo modelo de programação paralela e um conjunto de instruções, que permitem que aplicações paralelas sejam executadas em GPUs (*Graphics Processing Units*) para a solução de problemas complexos mais eficientemente que em CPU (*Central Unit Processing*).

Este poder de processamento está disponível aos programadores, porém desenvolver aplicações de propósito geral para GPU requer um bom conhecimento dos conceitos, ferramentas e capacidades que a tecnologia CUDA fornece.

CUDA permite que seus desenvolvedores utilizem C/C++ como linguagem de desenvolvimento. São fornecidas bibliotecas e alguns exemplos de aplicações pertencentes ao seu *kit* de desenvolvimento (SDK). Além de sua API padrão, suporta outras linguagens como FORTRAN, OpenCL (KHRONOS, 2010) e DirectCompute (MICROSOFT, 2010) (NVIDIA, 2012) e abordagens baseadas em diretivas para aceleradores como OpenACC (OPENACC, 2011).

7.6.1. Organização do Modelo de Programação

O conceito fundamental da execução paralela em dispositivos CUDA está associado ao uso de *threads* para o paralelismo de dados, de granularidade fina. Explora o paralelismo de dados existente para acelerar a execução das aplicações, pois uma grande quantidade de operações podem ser feitas sobre a estrutura de dados do programa simultaneamente (Kirk, 2010).

O modelo de programação CUDA assume que suas *threads* são executadas em um dispositivo separado, uma GPU, que trabalha como coprocessador do *host*. Desta maneira, o código principal é executado na CPU, e faz chamadas a funções que são executadas pela GPU, conforme apresentado na Figura 7.19.

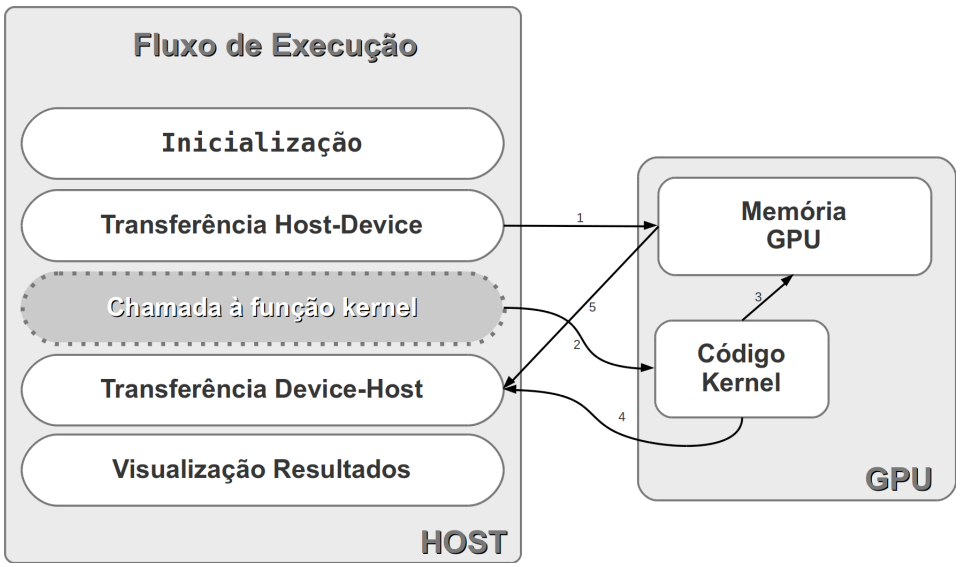

Figura 7.19: Fluxo de execução

As *threads* executam todas um mesmo código definido em uma função denominada *kernel*. Um *kernel* é um conceito estendido de uma função C, porém, ao contrário desta, quando uma chamada a uma função *kernel* é feita, são executadas *N* instâncias paralelas por *N threads* CUDA.

Uma função é definida como *kernel* usando-se a declaração **__global__** e uma chamada a uma função *kernel* distingue-se de uma chamada a uma função comum pela especificação do número de *threads* que a executarão. A especificação é dada usando a notação <<<…>>> na chamada. Os valores dentro dos <<<…>>> são parâmetros para o sistema de *runtime* que descrevem como deve ser a invocação da função *kernel*. O primeiro número indica o número de blocos paralelos e o segundo indica o número de *threads* por bloco. As *threads* em execução recebem um identificador disponível na variável *threadIdx* e acessível dentro do código do *kernel*, possibilitando o controle de acesso aos dados por determinadas *threads*.

A Figura 7.20 é um exemplo de código, originalmente disponível em (NVIDIA, 2012), que representa a definição de uma função *kernel* para a soma de dois vetores A e B, armazenando os resultados em um vetor C.

```
01 // Definição da Função Kernel.
02 __global__ void VecAdd(float* A, float* B, float* C) {
03     int i = threadIdx.x;
04     C[i] = A[i] + B[i];
05 }
06 int main() { ... // Invocação do Kernel com N threads.
07   VecAdd<<<1, N>>>(A, B, C);
08 …
09 }
```

Figura 7.20: Exemplo de *kernel* (Fonte: NVIDIA, 2012a)

O modelo de programação CUDA utiliza-se de uma estrutura, ou forma de organizar o contexto de execução, em: *grids*, blocos e *threads*, formando uma hierarquia de *threads*. Neste modelo, as *threads* são agrupadas em blocos e estes são agrupados em um *grid*, o que resulta em uma função *kernel* sendo executada como um *grid* de blocos de *threads*. Essa hierarquia de *threads* é mapeada para a hierarquia de processadores na GPU. As GPUs atuais executam um ou mais *grids* de *kernels*, enquanto um *streaming multiprocessor* (SM na Fermi ou SMX na Kepler) executa um ou mais blocos de *threads* e os *cores* e outras unidades de execução no SMX executam as instruções da *thread* (NVIDIA Corporation, 2012b).

Variáveis embutidas (**blockId.x**, **blockId.y**, **threadId.x** e **threadId.y**, por exemplo) são definidas pelo *runtime* do CUDA para possibilitar a manipulação do arranjo estrutural de *thread*s montado com a chamada à função *kernel*. Estes identificadores estão disponíveis durante a execução, possibilitando tanto o chaveamento no código quanto que decisões sejam tomadas, podendo o fluxo de execução ser alterado com base nos *ids* das *threads*.

É possível identificar qual bloco está sendo executado correntemente no dispositivo. CUDA permite definirmos um grupo de blocos, em até três dimensões, para problemas com domínios de dimensões variadas, tal como uma matriz ou uma imagem (bidimensionais, nestes casos). Quando lançamos a execução de um *kernel*, especificamos um número de blocos paralelos. Esse conjunto de blocos paralelos é chamado de *grid*.

Percebe-se que este não é um modelo trivial de programação, portanto será apresentado através de exemplos de código e figuras demonstrando a acomodação das *threads* em diferentes dimensões.

Supondo que quiséssemos adicionar um valor passado por parâmetro, a cada elemento de um arranjo de tamanho N, poderíamos utilizar a função *kernel* definida na Figura 7.21.

```
01 // Definição da função kernel.
02 __global__ void addValue(float *vector, float value, int n){
03   // Recupera o Id global da thread.
04   int id = blockIdx.x * blockDim.x + threadIdx.x;
05   // Garantia de que o id está nos limites do arranjo.
06   if (id < n)
07     vector[id] += value;
08 }
```

Figura 7.21: Exemplo de organização do arranjo de threads

A `linha 04` recupera o id da *thread* dentro do arranjo formado pelo *grid* e pelos blocos de *threads*, e na `linha 07` o valor é adicionado ao elemento do arranjo que é assistido pela *thread* .

No código principal, com as chamadas à função *kernel* exemplificadas na Figura 7.22 é possível perceber que existem pelo menos duas formas de estruturar o arranjo de *threads* e blocos. Na primeira forma (`linha 02`), especificamos ao *runtime* um *grid*

unidimensional com N blocos, e cada bloco com uma *thread*, na segunda (`linha 03`) declaramos um *grid* com 1 bloco com *N threads*.

```
01 // Restante do código suprimido.
02 addValue<<<N, 1>>>(d_vector, value, N);
03 addValue<<<1, N>>>(d_vector, value, N);
```

Figura 7.22: Chamadas à função *kernel*

Uma chamada a uma função *kernel* cria um *grid* de blocos de *threads*, as quais executam o código. Os identificadores **threadIdx.x** e **threadIdx.y** são identificadores associados a cada *thread* dentro de um bloco. Na chamada da `linha 02` especificamos um *grid* com N blocos, cada bloco com uma *thread*, desta forma os valores de `blockIdx.x` irão de 0 a N-1. Se N for igual a 4, por exemplo, os blocos são referenciados conforme a Figura 7.23.

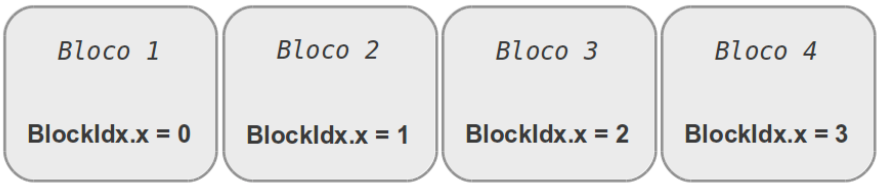

Figura 7.23: Bloco e seus identificadores

Uma forma mais genérica de especificar o arranjo de *threads* é apresentada na Figura 7.24, calculando-se o tamanho do grid em função do tamanho de bloco. Neste trecho de código o arranjo de *threads*, será preparado para trabalhar sobre uma estrutura de 64 elementos. A variável *gridSize*, na `linha 08`, irá receber o valor 8 pelo resultado de 64/8.

```
01 #define N 64
02 // Restante do código suprimido.
03 int blockSize, gridSize;
04 // Número de threads em cada bloco.
05 blockSize = 8;
06
07 // Número de blocos de threads no grid.
08 gridSize = (int)ceil((float)N/blockSize);
09
10 // Execute the kernel
11 addValue<<<gridSize, blockSize>>>(d_vector, value, N);
```

Figura 7.24: Chamadas à função *kernel*

A chamada à função *kernel* (`linha 11`) será `addValue<<<8,8>>>(d_vector, value, N)`. Assim teremos um *grid* formado por 8 blocos de 8 *threads* cada, conforme a Figura 7.25.

Figura 7.25: Disposição de 64 *threads* em um grid (8x8)

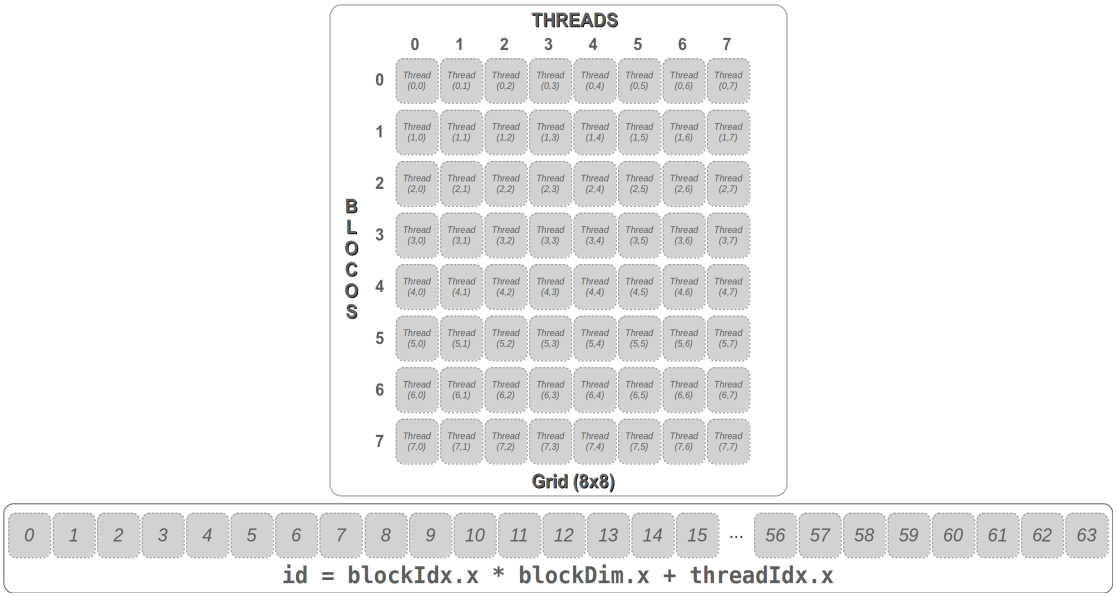

Figura 7.26: Tradução para endereçamento linear

As dimensões para um *grid* são limitadas a 65535 x 65535 x 1, bem como o número de *threads* em um bloco não pode ser excedido, o campo `maxThreadsPerBlock` do registro de informações do dispositivo que pode ser

recuperado chamando a função `cudaGetDeviceProperties(...)`. Para muitos dispositivos esse número é de 512 *threads* por bloco. Para problemas que envolvam estruturas de dados (vetores e matrizes) que ultrapassem essas dimensões, pode-se utilizar uma combinação de blocos e *threads* em um *grid*. Com um arranjo estrutural de computação com múltiplos blocos e múltiplas *threads*, a indexação será análoga ao método para conversões entre espaço bidimensional e o espaço linear, conforme apresentado na Figura 7.26.

7.6.2. Gerenciamento de Memória

Como GPU e CPU possuem memórias separadas, a memória principal é espaço de endereçamento da CPU e a GPU possui sua própria memória. É possível gerenciar a memória da GPU pelo código em execução no *host* (CPU), funções similares às funções já conhecidas dos programadores em linguagem C são fornecidas para que a alocação e liberação de memória do dispositivo possa ser feita. É possível também copiar dados do *host* para o dispositivo, do dispositivo para o *host* e de dispositivo para dispositivo.

7.6.2.1 Alocação de memória na GPU

Regiões de memória destinadas a variáveis ou estruturas de dados podem ser alocadas no espaço de endereçamento da GPU usando-se a função `cudaMalloc()`. E podem ser inicializadas com um valor padrão usando-se a função `cudaMemset()`, após o uso essa memória pode ser liberada com chamadas à função `cudaFree()`. A transferência de dados entre as memórias do *host* e do dispositivo são feitas utilizando-se a função `cudaMemcpy()`, a direção da cópia é dada pelos tipos: `cudaMemcpyHostToHost` (*host* para *host*), `cudaMemcpyHostToDevice` (*host* para dispositivo), `cudaMemcpyDeviceToHost` (dispositivo para *host*) e `cudaMemcpyDeviceToDevice` (dispositivo para dispositivo). A Figura 7.27 apresenta as funções de manipulação de memória.

```
cudaMalloc(void **pointer, size_t nbytes)
cudaMemset(void *pointer, int value, size_t count)
cudaMemcpy(void *dst, const void *src, size_t count, enum
cudaMemcpyKind kind)
cudaFree(void *pointer)
```

Figura 7.27: Funções de Manipulação de Memória

A Figura 7.28 na linha 05 exemplifica a alocação de memória para um arranjo no *host*, as linhas 06 e 07 fazem a alocação e inicialização da estrutura na memória da GPU. No mesmo trecho de código é feita cópia do arranjo da memória do host para a memória da GPU, e no final as funções de liberação de memória, tanto na GPU (linha 11), quanto no host (linha 12).

```
01 int n = 1024;
02 int nbytes = 1024 * sizeof(int);
03 int *h_a = 0;
04 int *d_a = 0;
05 h_a = (int*) malloc(nbytes); // Alocação de Memória no host.
```

```
06 cudaMalloc((void**)&d_a, nbytes); // Alocação de Memória da GPU.
07 cudaMemset( d_a, 0, nbytes); // Inicialização da Memória alocada
08 na GPU.
09 cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice); // Cópia
10 host -> GPU.
11 …
12 cudaFree(d_a); // Liberação de Memória na GPU.
   free(h_a); // Liberação de Memória no host.
```

Figura 7.28: Alocação e cópia de um *array* para a GPU

7.6.3. Exemplo: Soma de Vetores

Seguindo os conceitos apresentados nas seções anteriores, a Figura 7.29 traz um exemplo completo, uma versão para CUDA do algoritmos de soma de vetores. Este exemplo é baseado em um tutorial oferecido pelo OLCF (*Oak Ridge Leadership Computing Facility*), um dos maiores centros de processamento de alto desempenho do mundo (OLCF, 2012).

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <math.h>
04
05 // CUDA kernel. Cada thread é responsável por um elemento de c.
06 __global__ void vecAdd(float *a, float *b, float *c, int n)
07 {
08     // Recupera o ID global da thread.
09     int id = blockIdx.x*blockDim.x+threadIdx.x;
10
11     // Verificação se o id está dentro dos limites.
12     if (id < n)
13         c[id] = a[id] + b[id];
14 }
15
16 int main( int argc, char* argv[] )
17 {
18     // Tamanho dos vetores.
19     int n = 100000;
20
21     // Declaração do vetores de entrada do Host.
22     float *h_a;
23     float *h_b;
24     // Declaração do vetor de saída do Host.
25     float *h_c;
26
27     // Declaração dos vetores de entrada na memória da GPU.
28     float *d_a;
29     float *d_b;
30     // Declaração do vetor de saída do dispositivo.
31     float *d_c;
32
33     // Tamanho em bytes de cada vetor.
34     size_t bytes = n*sizeof(float);
35
36     // Alocação de memória para os vetores do host.
37     h_a = (float*)malloc(bytes);
38     h_b = (float*)malloc(bytes);
```

```
39      h_c = (float*)malloc(bytes);
40
41      // Alocação de memória para cada vetor na GPU.
42      cudaMalloc(&d_a, bytes);
43      cudaMalloc(&d_b, bytes);
44      cudaMalloc(&d_c, bytes);
45
46      int i;
47      // Inicialização dos vetores do host.
48      for( i = 0; i < n; i++ ) {
49          h_a[i] = sinf(i)*sinf(i);
50          h_b[i] = cosf(i)*cosf(i);
51      }
52
53      // Cópia dos vetores do host para o dispositivo.
54      cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
55      cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
56
57      int blockSize, gridSize;
58
59      // Número de threads em cada bloco de threads.
60      blockSize = 1024;
61
62      // Número de blocos de threads no grid.
63      gridSize = (int)ceil((float)n/blockSize);
64
65      // Chamada à função kernel.
66      vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
67
68      // Cópia do vetor resultado da GPU para o host.
69      cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );
70
71      // Soma do vetor c e impressão do resultado dividido por n,
72  que deve ser igual a 1.
73      float sum = 0;
74      for(i=0; i<n; i++)
75          sum += h_c[i];
76      printf("Resultado Final: %f\n", sum/n);
77      // Liberação da memória da GPU.
78      cudaFree(d_a);
79      cudaFree(d_b);
80      cudaFree(d_c);
81
82      // Liberação da memória do host.
83      free(h_a);
84      free(h_b);
85      free(h_c);
86
87      return 0;
88  }
```

Figura 7.29: Soma de Vetores. Fonte: (OLCF, 2012)

7.7. OpenCL: biblioteca de programação heterogênea

OpenCL (Munchi, 2011) possui uma filosofia ligeiramente diferente de CUDA. Em OpenCL a linguagem e seu sistema de tempo de execução servem como uma camada de abstração ao hardware heterogêneo. Assim, um programa OpenCL tem o objetivo de aproveitar todos os dispositivos presentes na máquina, tais como processadores

multicore, GPUs, DSPs (*Digital Signal Processors*), entre outros. Este capítulo apresenta rapidamente a linguagem e seu ambiente de execução e é baseado no texto apresentado em (Silva, 2012). Uma aplicação OpenCL que executa num hardware heterogêneo deve seguir os seguintes passos:

- Descobrir os componentes que compõem o sistema heterogêneo;
- Detectar as características do hardware heterogêneo tal que a aplicação possa se adaptar a elas;
- Criar os blocos de instruções (*kernels*) que irão executar na plataforma heterogênea;
- Iniciar e manipular objetos de memória;
- Executar os *kernels* na ordem correta e nos dispositivos adequados presentes no sistema;
- Coletar os resultados finais.

Estes passos podem ser realizados através de uma série de APIs do OpenCL juntamente com um ambiente de programação e execução dos *kernels*. Esta seção apresenta um resumo do modelo de abstração do OpenCL juntamente com um exemplo simples de código. O modelo de abstração é apresentado através de três modelos fundamentais descritos a seguir: modelo de plataforma, modelo de execução e modelo de memória.

7.7.1. Modelo de plataforma

O modelo de plataforma é composto por um *host* e um ou mais dispositivos OpenCL (*OpenCL devices*). Cada dispositivo possui uma ou mais unidades de computação (*compute units*), que por sua vez são compostos por um conjunto de elementos de processamento (*processing elements*). A Figura 7.30 apresenta esta organização.

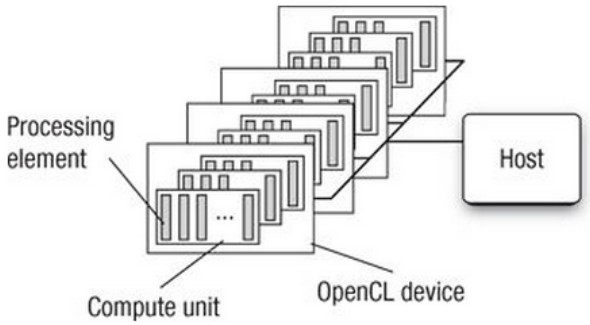

Figura 7.30: modelo de plataforma do OpenCL (Fonte: Munchi, 2011)

O *host* é conectado a um ou mais dispositivos e é responsável pela inicialização e envio dos *kernels* para a execução nos dispositivos heterogêneos. Os dispositivos normalmente são compostos por unidades de computação que agrupam uma determinada quantidade de elementos de processamento. Em relação à CUDA, as unidades de computação correspondem aos *Streaming Multiprocessors* da GPU (dispositivo) e os elementos de processamento correspondem aos núcleos (*cores*). Um

dispositivo, portanto, pode ser uma CPU, GPU, DSP ou outro qualquer, dependendo da implementação do OpenCL.

7.7.2. Modelo de execução

O modelo de execução define que uma aplicação OpenCL é composta por um programa *host* e um conjunto de *kernels*. O programa *host* executa no *host* (normalmente uma CPU) e os *kernels* executam nos dispositivos disponíveis.

O programa *host*, ou simplesmente *host*, envia o comando de execução de um *kernel* para um dispositivo. Isto faz com que várias instâncias da função que implementa o *kernel* sejam executadas no dispositivo alvo. Em OpenCL estas instâncias são chamadas de itens de trabalho (*work-items*) e correspondem às *threads* de CUDA. Assim como em CUDA, cada *thread* ou item de trabalho é identificado por suas coordenadas no espaço de índices (que aqui também pode ter até 3 dimensões) e correspondem ao seu *global ID*.

Os itens de trabalho são organizados, por sua vez, em grupos de trabalho (*work-groups*). Estes oferecem uma maneira de estabelecer granularidades diferentes aos grupos de itens de trabalho, o que normalmente facilita a divisão de trabalho e a sincronização. Os grupos de trabalho correspondem aos blocos de CUDA e podem ser situados num espaço de até três dimensões. Assim, os itens de trabalho possuem dois tipos de coordenadas: local (dentro do grupo de trabalho) e global (dentro do conjunto completo de itens de trabalho em execução).

O *host* deve ainda definir um contexto (*context*) para a aplicação OpenCL. Um contexto define o ambiente de execução no qual os *kernels* são definidos e executam e é definido em termos dos seguintes recursos: dispositivos, conjunto de *kernels*, objetos de programa (códigos fonte e executável dos *kernels* que executam a aplicação) e objetos de memória (dados que serão utilizados pelos *kernels* durante o processamento). Assim, um contexto é todo o conjunto de recursos que um *kernel* vai utilizar durante sua execução.

O contexto é definido em tempo de execução pelo *host* de acordo com os dispositivos disponíveis na máquina alvo. Para possibilitar uma escolha dinâmica do dispositivo onde os *kernels* vão executar o OpenCL compila os *kernels* dinamicamente, gerando os objetos de programa, portanto, em tempo de execução.

A interação entre o *host* e os dispositivos é realizada através de uma fila de comandos (*command-queue*). Os comandos são colocados nesta fila e aguardam seu momento de executar. A fila é criada pelo *host* e conectada a um dispositivo logo após a criação do contexto. Esta fila aceita três tipos de comandos: execução de *kernel*, transferência de dados (objetos de memória) e comandos de sincronização.

Os comandos colocados em uma fila executam de forma assíncrona com relação ao *host*. Comandos de sincronização podem ser utilizados caso uma ordem deva ser estabelecida. Os comandos na fila normalmente executam em ordem (*in-order execution*), porém algumas implementações de OpenCL podem oferecer o modo de execução fora de ordem (*out-of-order execution*), que prevê uma execução assíncrona dos comandos enfileirados.

7.7.3. Modelo de memória

O modelo de memória do OpenCL define dois tipos de objetos de memória: *buffers* (blocos contíguos de memória aos quais é possível mapear estruturas de dados) e imagens. Estas podem ser manipuladas através de funções específicas presentes na API do OpenCL.

O modelo de memória define cinco regiões diferentes (Figura 7.31):

- Memória do *host* (*host memory*): visível apenas ao *host*;
- Memória global (*global memory*): permite acesso para leitura e escrita a partir de todos os itens de trabalho em todos os grupos de trabalho;
- Memória constante (*constant memory*): é uma memória global que é inicializada pelo *host* e permite acesso somente de leitura aos itens de trabalho;
- Memória local (*local memory*): é compartilhada apenas entre os itens de trabalho de um mesmo grupo de trabalho;
- Memória privada (*private memory)*: fica no escopo de cada item de trabalho.

A interação entre o *host* e o modelo de memória pode ocorrer de duas maneiras: cópia explícita ou mapeamento de regiões de um objeto de memória. Na cópia explícita, comandos de transferência entre *host* e dispositivos são enfileirados na fila de comandos e podem ser executados de forma síncrona ou assíncrona. No método de mapeamento, os objetos de memória são mapeados na memória do *host*, que pode também realizar acessos a estes objetos. O comando de mapeamento também deve ser enfileirado na fila de comandos.

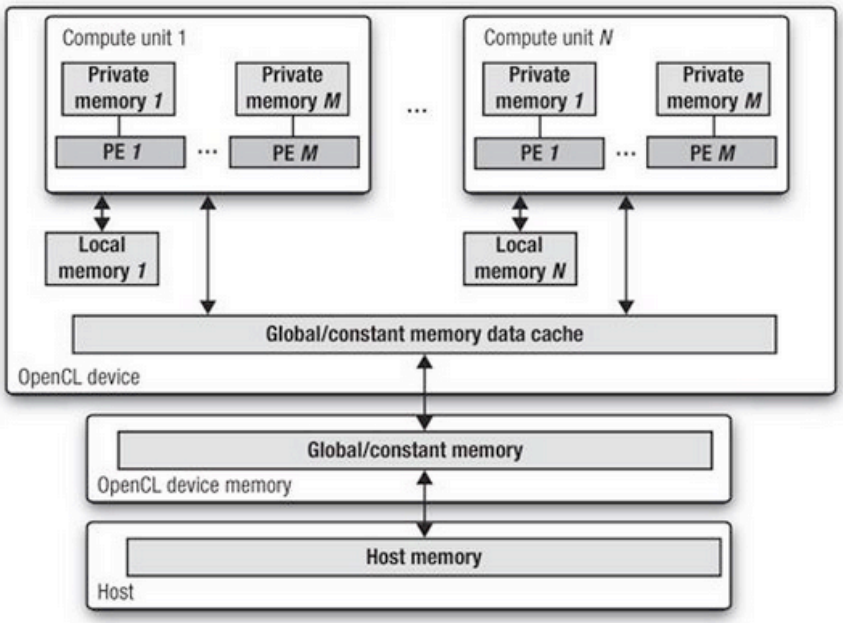

Figura 7.31: Regiões de memória de OpenCL (Fonte: Munchi, 2011)

7.7.4. Exemplo: soma de vetores em OpenCL

As Figuras 7.32 e 7.33 apresentam um exemplo de soma de vetores em OpenCL. Este exemplo é baseado em um tutorial oferecido pelo OLCF (*Oak Ridge Leadership Computing Facility*), um dos maiores centros de processamento de alto desempenho do mundo (OLCF, 2012).

A Figura 7.32 apresenta o código do *kernel*, que pode ficar num arquivo separado (.cl) ou pode ser formatado no próprio código como uma *string-C*. Este código será passado como argumento à função OpenCL que o compilará em tempo de execução.

A partir do código da Figura 7.32 é possível observar algumas características de OpenCL:

- A definição de um *kernel* é feita através de uma função que utiliza o modificador __kernel (linha 01);
- O modificador __global indica que os parâmetros estão na memória global do dispositivo (linhas 01 a 03);
- A função get_global_id() devolve o identificador global da *thread* (*work item*) na dimensão 0 (linha 06);
- A verificação do identificador (linha 07) é comumente realizada neste tipo de computação, pois por motivos de desempenho é possível que *threads* a mais venham a ser lançadas. A verificação serve para que somente as *threads* "dentro do problema" executem o trabalho. Este tipo de verificação também é comum em CUDA;
- Na linha 08 a soma é realizada (*n threads* serão iniciadas e cada uma realizará uma soma).

```
01  __kernel void vecAdd(__global const float *a,
02                        __global const float *b,
03                        __global float *c,
04                        const unsigned int n)
05  {
06      int gid = get_global_id(0);
07      if (gid < n)
08         c[gid] = a[gid] + b[gid];
09  }
```

Figura 7.32: *Kernel* em OpenCL (vecAdd.cl)

A Figura 7.33 apresenta a *main()* juntamente com funções auxiliares do OpenCL que devem ser executadas pelo *host*. Para reduzir o tamanho do código algumas partes foram omitidas e estão indicadas em comentários. O código completo pode ser encontrado em (OLCF, 2012).

```
01 #include <CL/opencl.h>
02
03 // Demais includes
04
05 ...
06 int main( int argc, char* argv[] )
07 {
08     // Declarações e inicializações
09     ...(omitido)
10     // Alocação de memória inicialização para cada vetor no host
11     ...(omitido)
12     // Número de itens de trabalho em cada grupo de trabalho local
13     size_t localSize = 64;
14     // Número total de itens de trabalho
15     size_t globalSize = ceil(n/(float)localSize)*localSize;
16
17     /* Obtém a plataforma, obtém o ID do dispositivo (GPU), cria um
18 contexto, cria uma command queue, lê o código fonte do kernel e transforma
19 numa string-C */
20     ...(omitido)
21
22     // Cria o programa
23     program = clCreateProgramWithSource(context, 1,
24                     (const char **) &kernelSource, NULL, &err);
25     // Compila o executável
26     clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
27     // Cria o kernel
28     kernel = clCreateKernel(program, "vecAdd", &err);
29     // Cria os arrays de entrada e saída na memória do dispositivo
30    d_a =clCreateBuffer(context,CL_MEM_READ_ONLY,bytes,NULL,NULL);
31    d_b =clCreateBuffer(context,CL_MEM_READ_ONLY,bytes,NULL,NULL);
32    d_c =clCreateBuffer(context,CL_MEM_WRITE_ONLY,bytes,NULL,NULL);
33     // Escreve os dados no array de entrada do dispositivo
34     err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
35                                   bytes, h_a, 0, NULL, NULL);
36     err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
37                                   bytes, h_b, 0, NULL, NULL);
38     // Envia argumentos ao kernel
39     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
40     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
41     err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
42     err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
43     // Executa o kernel em todo o intervalo de dados
44     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL,
45                         &globalSize, &localSize,0, NULL, NULL);
46     // Espera que finalização do kernel antes de ler os resultados
47     clFinish(queue);
48     // Lê resultados a partir do dispositivo
49     clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
50                                 bytes, h_c, 0, NULL, NULL );
51     //Teste: Soma o vetor c e imprime o resultado dividido por n    //
52 libera recursos do OpenCL e a memória do host
53     ...(omitido)
54     return 0;
}
```

Figura 7.33: Soma de vetores em OpenCL (Fonte: OLCF, 2012)

A seguir, destacamos as principais características de OpenCL presentes no código:

- Na linha 12 é definido o *localSize* que é a quantidade de itens de trabalho em cada grupo de trabalho – 64 neste caso. Isto equivale em CUDA a definir a quantidade de *threads* em cada bloco;
- A linha 14 define a quantidade total de itens de trabalho lançados (*globalSize*). Num primeiro momento pensaríamos que este número deve ser igual ao tamanho do vetor (*n*). Porém, *globalSize* deve ser divisível por *localSize*, por isso o arredondamento realizado nesta linha;
- A partir da linha 16 deve ser realizado o *setup* do OpenCL (omitido por motivo de espaço): plataforma, dispositivo, contexto e fila de comandos (*command queue*);
- Entre as linhas 21 e 27 o *kernel* e compilado e criado;
- Entre as linhas 29 e 41 os dados são preparados e enviados ao dispositivo;
- Na linha 43 o *kernel* é enfileirado e por fim lançado no dispositivo;
- Na linha 46 o *host* espera a finalização do *kernel* (sincronização);
- Na linha 48 o resultado é lido da memória do dispositivo.

7.8. OpenACC: diretivas de programação para GPU

O objetivo desta seção é descrever as principais diretivas do OpenACC, destacando a possibilidade de heterogeneidade caso ele seja adicionado ao OpenMP devido às suas semelhanças intrínsecas.

O OpenACC é um padrão para programação paralela que foi anunciado em novembro de 2011 na conferência SuperComputing 2011, em uma parceria entre NVIDIA, Cray Inc., Portland Group (PGI), e CAPS Enterprise. O padrão tem como base o compilador PGI desenvolvido pelo Portland Group que descreve uma API de programação que fornece uma coleção de diretivas para especificar laços e regiões de código paralelizáveis que podem ter sua execução acelerada por dispositivo tal como uma GPU (OPENACC, 2011) (OPENACC STANDARD, 2011).

No OpenACC as transferências de dados entre as memórias do dispositivo acelerador e do *host* e *caching* de dados são implícitos e gerenciados pelo compilador com base em anotações feitas pelo programador por meio de diretivas do OpenACC.

Percebe-se rapidamente as vantagens de se usar o modelo de diretivas em relação ao modelo tradicional de CUDA. O padrão ainda não está implementado, mas espera-se uma primeira versão de compilador para meados de 2012.

O modelo de execução do OpenACC tem três níveis: ***gang***, ***worker*** e ***vector***. Esta estruturação é para ser mapeada em uma arquitetura organização em uma coleção de elementos de processamento. Cada um destes elementos é *multithread* e cada *thread* pode executar instruções vetoriais. No contexto de GPUs um mapeamento possível poderia ser *gang=bloco*, *worker=warp*, *vector=threads em um warp* (um *warp* é um conjunto de *threads* escalonáveis num multiprocessador (SM)).

As diretivas OpenACC em C/C++ são especificadas usando diretivas `#pragma`,

que é um método especificado pelo C padrão para fornecer informações adicionais ao compilador (GCC, 2012), torna-se uma alternativa interessante, pois caso o compilador não reconheça ou não esteja preparado para utilizar esse mecanismo de pré-processamento, essas anotações serão ignoradas na compilação do código.

7.8.1. Diretivas

Cada diretiva em C/C++ é sensível ao caso e inicia com `#pragma acc`, conforme a Figura 7.34, os *tokens* seguintes são tratados por uma macro de pré-processamento que aplica as transformações necessárias à declaração imediatamente seguinte, podendo esta ser um bloco ou um laço.

```
01  #pragma acc directive-name [clause [[,] clause]…] new-line
```

Figura 7.34: Formato da diretiva OpenACC

Mais detalhes sobre as diretivas do OpenACC e das cláusulas aceitas com cada construção, podem ser obtidos consultando-se o manual disponível no sítio do padrão OpenACC (OPENACC STANDARD, 2011). As construções aplicáveis como nomes de diretivas são listadas na Tabela 7.3.

A Figura 7.35 apresenta um exemplo de utilização da diretiva **parallel** na paralelização de um laço no cálculo do valor do número PI. Na linha 06 a construção **parallel** é combinada com a construção **loop**. Algumas combinações entre as construções são possíveis, neste caso a combinação é um atalho para a declaração de uma diretiva **loop** dentro de uma região paralela e afeta o laço seguinte.

```
01  #include <stdio.h>
02  #define N 1000000
03
04  int main(void) {
05    double pi = 0.0f; long i;
06    #pragma acc parallel loop
07    for (i=0; i<N; i++) {
08       double t= (double)((i+0.5)/N);
09       pi +=4.0/(1.0+t*t);
10    }
11    printf("pi=%16.15f\n",pi/N);
12    return 0;
13  }
```

Figura 7.35: Exemplo Cálculo do PI (Fonte: OPENACC, 2012)

Tabela 7.3: Construções para diretivas OpenACC (Fonte: OPENACC, 2011)

Construção	Formato	Significado
parallel	`#pragma acc parallel [clause [[,]clause]…] new-line` `structured block`	Blocos (*gangs*) de *threads* (*workers*) são criados para executar em paralelo a região demarcada. Uma t*hread* em cada bloco inicia a execução do código `structured block`.
kernels	`#pragma acc kernels [clause [[,] clause]…] new-line` `structured block`	Define uma região que pode ser compilada e transformada em uma sequência de *kernels*. Tipicamente, cada laço pode ser um *kernel* distinto, com suas configurações de blocos e *threads*.
data	`#pragma acc data [clause [[,] clause]…] new-line` `structured block`	Define variáveis escalares, *arrays* e *subarrays* para serem alocados na memória da GPU durante a execução da região, define ainda se os dados devem ser copiados do host para o dispositivo e se ao término da execução da região, devem ser copiados de volta.
loop	`#pragma acc loop [clause [[,] clause]...]new-line for loop`	Descreve que tipo de paralelismo deve ser usado na execução do laço, pode declarar variáveis privadas, operações de redução e se as iterações são independentes.
cache	`#pragma acc cache( list ) new-line`	Especifica elementos de um *array* ou um *subarray* que devem ser mantidos na cache, por exemplo o corpo de um laço.
update	`#pragma acc update clause [[,] clause]... new-line`	Utilizada para atualizar a mémoria do *host* copiando os dados do dispositivo e vice-versa.
declare	`#pragma acc declare declclause [[,] declclause]... new-line`	Pode especificar variáveis ou *arrays* que devem ser alocados na GPU durante a região de dados de uma função.
wait	`#pragma acc wait [( scalar-integer-expression )] new-line`	Faz com que o programa aguarde o término de uma atividade assíncrona, tal como uma região de *kernels* ou a execução de uma diretiva de *update*.

A saída produzida pelo processo de compilação do código da Figura 7.35 e o resultado produzido pela execução do programa são apresentadas na Figura 7.36.

```
$ pgcc -acc -ta=nvidia -Minfo=accel -fast pi.c -o pi
main:
 6, Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
 7, Loop is parallelizable
    #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
    CC 1.3 : 21 registers; 2072 shared, 36 constant, 0 local memory
bytes; 50% occupancy
    CC 2.0 : 19 registers; 2056 shared, 44 constant, 0 local memory
```

```
bytes;100% occupancy
 9, Sum reduction generated for pi

$ ./pi
pi=3.141592653589877
```

Figura 7.36: Saída da compilação e execução do exemplo PI

O código da Figura 7.37 é uma versão modificada de um exemplo (PGI OpenACC, 2012) para a soma de dois vetores. Nota-se que as únicas linhas adicionadas a uma versão sequencial que executaria totalmente no host são as anotações OpenACC que estão nas linhas 12 e 44. O código anotado não exclui a possibilidade de compilação em um compilador comum, tal como o gcc, pois as diretivas serão pré-processadas, mas não produzirão efeito algum.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  /* Assinatura das funções (omitidas) */
05  void inicializar(float* vetor, int n);
06  void imprimir(float* vetor, int n);
07  void vecaddhost(float *e, float *a, float *b, int n);
08  int comparar(float *r, float *e, int n);
09
10  /* Função que será transformada em kernel */
11  void vecaddgpu(float *restrict r, float *a, float *b, int n){
12    #pragma acc kernels for present(r,a,b)
13    for( int i = 0; i < n; ++i )
14      r[i] = a[i] + b[i];
15  }
16
17  /* Programa principal */
18  int main( int argc, char* argv[] ){
19    int n; /* tamanho do vetor */
20    float * a; /* declaração do vetor 1 */
21    float * b; /* declaração do vetor 2 */
22    float * r; /* declaração do vetor de resultado */
23    float * e; /* declaração do vetor de verificação */
24
25    int i, erros;
26
27    // Define a quantidade de elementos, parâmetro ou valor
28  default.
29    if( argc > 1 )
30      n = atoi( argv[1] );
31    else n = 100000; /* tamanho padrão */
32    if( n <= 0 ) n = 100000;
33
34    // Alocação dos vetores.
35    a = (float*)malloc( n*sizeof(float) );
36    b = (float*)malloc( n*sizeof(float) );
37    r = (float*)malloc( n*sizeof(float) );
38    e = (float*)malloc( n*sizeof(float) );
39
40    // Inicialização dos vetores de entrada.
41    inicializar(a, n);
42    inicializar(b, n);
```

```
43
44    /* Execução na GPU */
45    #pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
46    {
47      vecaddgpu(r, a, b, n);
48    }
49
50    /* cálculo do resultado para comparação */
51    vecaddhost(e, a, b, n);
52
53    /* comparar os resultados */
54    erros = 0;
55    erros = comparar(r, e, n);
56
57    /* imprimir os vetores */
58    imprimir(a, n);
59    imprimir(b, n);
60    imprimir(r, n);
61    imprimir(e, n);
62
63    printf("Resultado: %d erros encontrados.\n", erros );
64    return erros;
65  }
66
67  void vecaddhost(float *e, float *a, float *b, int n){
68    for( int i = 0; i < n; ++i )
69      e[i] = a[i] + b[i];
70  }
71
72  int comparar(float *r, float *e, int n){
73    int i, erros = 0;
74    for( i = 0; i < n; ++i ){
75      if( r[i] != e[i] ){
76        ++erros;
77      }
78    }
79    return erros;
80  }
81
82
83
```

Figura 7.37: Exemplo Soma de Vetores com diretivas OpenACC

A diretiva **acc** data da linha 44, que envolve a chamada à função **vecaddgpu** no programa principal, faz com que os dados, no caso vetores (**a, b**) sejam copiados para a memória da GPU e o vetor resultado (**r**) seja copiado de volta para a memória do *host* ao final da execução da região paralela. A cláusula **present** na função **vecaddgpu** diz ao compilador que os dados já estão presentes na GPU, que foram copiados anteriormente. O resultado da compilação é apresentado na Figura 7.38.

A saída apresentada na Figura 38 mostra as detecções que o compilador **pgcc** fez em relação às diretivas inseridas no código. Na linha 12 detectou a cláusula **present** que gerou duas versões de código, um para dispositivos com capacidade de computação 1.0 e outro com 2.0 (capacidade de computação ou *compute capability* faz um mapeamento entre a versão do hardware da placa e a plataforma CUDA). Na linha 13 identificou o laço for como paralelizável e também mostra o escalonamento usado para o laço (***gang***,

vector(256)), que suas iterações serão quebradas em blocos de 256 *threads* que serão executadas em paralelo.

```
$ pgcc -acc -ta=nvidia,time -Minfo=accel -fast vectoradd-3.c -o
vectoradd-3-acc-gpu
vecaddgpu:
  12, Generating present(b[0:])
      Generating present(a[0:])
      Generating present(r[0:])
      Generating compute capability 1.0 binary
      Generating compute capability 2.0 binary
  13, Loop is parallelizable
      Accelerator kernel generated
      13,  #pragma  acc  loop  gang,  vector(256)  /*  blockIdx.x
threadIdx.x */
       CC 1.0 : 5 registers; 36 shared, 4 constant, 0 local memory
bytes; 100% occupancy
       CC 2.0 : 5 registers; 4 shared, 48 constant, 0 local memory
bytes; 100% occupancy
main:
  44, Generating copyout(r[0:n])
      Generating copyin(b[0:n])
      Generating copyin(a[0:n])
```

Figura 7.38: Saída da compilação do exemplo soma vetores

Analisar a saída do compilador é importante, pois quando não for possível paralelizar as regiões anotadas, o resultado mostrará dependências que impedem a paralelização e dirá que código de execução sequencial foi gerado (**#pragma acc loop seq**).

A ideia de anotar regiões e construções que devem ser paralelizadas, trazida pelo padrão OpenACC, e implementada pelo compilador do PGI, o **pgcc** mostra-se bem interessante, pois anotar o código não prejudica a compilação em compiladores não direcionados à aceleração, o código é compilado normalmente, sendo as diretivas ignoradas.

7.9. Programação heterogênea em outras linguagens

O objetivo desta seção é descrever os principais *bindings* e bibliotecas para acesso aos recursos de CUDA e de GPUs em outras linguagens, tais como Java, Python e C++/STL. A criação de *bindings* acontece quando existem bibliotecas e se deseja acessá-las em uma outra linguagem, mas sem ser necessário reescrevê-las. No nosso caso, acessar recursos das APIs CUDA (NVIDIA, 2012) e OpenCL (KHRONOS, 2011) nas linguagens Java e Python. Além disso, apresentamos também a biblioteca de algoritmos paralelos Thrust, que assemelha-se à *Standard Template Library* (STL) da linguagem C++ (Thrust, 2012).

7.9.1. JCuda e JOCL

O JCuda (JCUDA, 2012) e o JOCL (JOCL, 2012) são *bindings* feitos para linguagem Java, para acessar os recursos das bibliotecas do *runtime* e do *driver* CUDA e OpenCL. São APIs que são muito próximas das bibliotecas originais, exceto pelas restrições

específicas da linguagem Java, com relação a ponteiros por exemplo, neste caso uma classe **Pointer** é fornecida. A Figura 7.39 apresenta o exemplo da soma de vetores em código equivalente do JCuda. Nota-se que o código faz chamadas a funções do Driver através do JCuda, o código é bem parecido com o exemplo de utilização da API do Driver. O código PTX carregado pode ser de um arquivo existente, ou de um obtido pela execução do comando de compilação com o **nvcc** e a opção **--ptx**, por meio do *runtime* do Java. O código das funções de inicialização, impressão e verificação dos resultados foi omitido.

```
01  // imports ...(omitido)
02  public class JCudaVectorAdd {
03    public static void main(String args[]) throws IOException{
04      // Tamanho dos vetores.
05      int n = 1024;
06
07      // Alocação de memória no host para os vetores.
08      // Inicialização dos vetores.
09      // ...(omitido)
10
11      // Habilitar exceções.
12      JCudaDriver.setExceptionsEnabled(true);
13
14      // Nome do arquivo PTX a ser utilizado.
15      String ptxFileName = "vector_add_kernel.ptx";
16
17      // Inicialização da API do Driver.
18      cuInit(0);
19      // Define um handle para o dispositivo 0.
20      CUdevice device = new CUdevice();
21      cuDeviceGet(device, 0);
22
23      // Cria um contexto.
24      CUcontext context = new CUcontext();
25      cuCtxCreate(context, 0, device);
26
27      // Cria um módulo de um arquivo ptx.
28      CUmodule module = new CUmodule();
29      cuModuleLoad(module, ptxFileName);
30
31      // Alocação dos vetores na memória da GPU.
32      CUdeviceptr dA = new CUdeviceptr();
33      cuMemAlloc(dA, n * Sizeof.FLOAT);
34
35      CUdeviceptr dB = new CUdeviceptr();
36      cuMemAlloc(dB, n * Sizeof.FLOAT);
37
38      CUdeviceptr dC = new CUdeviceptr();
39      cuMemAlloc(dC, n * Sizeof.FLOAT);
40
41      // Cópia dos vetores da memória do host para a memória da
42  GPU.
43      cuMemcpyHtoD(dA, Pointer.to(hA), n * Sizeof.FLOAT);
44      cuMemcpyHtoD(dB, Pointer.to(hB), n * Sizeof.FLOAT);
45
46      // Cria um manipulador para a função kernel.
47      CUfunction function = new CUfunction();
48      cuModuleGetFunction(function, module, "vector_add_kernel");
49
```

```
50      // Definição dos arranjo de threads.
51      int threadsPerBlock = 16;
52    int blocksPerGrid =
53  (int)Math.ceil((double)(n       +       threadsPerBlock       -
54  1)/threadsPerBlock);
55
56      // Parâmetros para o kernel.
57      Pointer parametros = Pointer.to(
58        Pointer.to(dA),
59        Pointer.to(dB),
60        Pointer.to(dC),
61        Pointer.to(new int[]{n})
62      );
63      // Chamada ao kernel.
64      cuLaunchKernel(function,
65                     blocksPerGrid,  1, 1,  // Dimensão do grid.
66                     threadsPerBlock, 1, 1, // Dimensão do Bloco.
67                     0, null,  // Memoria compartilhada e a
68                     stream.parametros, null       // Parâmetros.
69      );
70      // Sincronização.
71      cuCtxSynchronize();
72
73      // Cópia do vetor resultado da GPU para o host.
        cuMemcpyDtoH(Pointer.to(hC), dC, n * Sizeof.FLOAT);

        // Impressão e verificação do resultado.
        // ...(omitido)

        // Liberação de Memória.
        cuMemFree(dA);
        cuMemFree(dB);
        cuMemFree(dC);
    }
```

Figura 7.39: Soma de Vetores em JCuda

7.9.2. PyCUDA e PyOpenCL

PyCUDA e PyOpenCL (PyCUDA, 2012) são *bindings* de CUDA e OpenCL para a linguagem Python. A Figura 7.40 apresenta o código do exemplo soma de vetores em PyCUDA.

```
01  import pycuda.autoinit
02  import pycuda.driver as drv
03  import numpy
04
05  from pycuda.compiler import SourceModule
06
07  mod = SourceModule("""
08  __global__ void vector_add_kernel(float *c, float *a, float *b){
09      const int id = blockIdx.x * blockDim.x + threadIdx.x;
10      c[id] = a[id] + b[id];
11  }
12  """)
13
14  add_func = mod.get_function("vector_add_kernel")
15  a = numpy.random.randn(1024).astype(numpy.float32)
```

```
16  b = numpy.random.randn(1024).astype(numpy.float32)
17
18  c = numpy.zeros_like(a)
19  add_func(
20          drv.Out(c), drv.In(a), drv.In(b),
21          block=(16,1,1), grid=(64,1))
22
23  print c-(a+b)
```

Figura 7.40: Soma de Vetores em PyCUDA

7.9.3. Thrust

Thrust é uma biblioteca de algoritmos paralelos que se assemelha à Standard Template Library de C++. É uma interface de alto nível com interoperabilidade com outros ambientes de programação tais como CUDA e OpenMP, entre outros.

Thrust é heterogênea e possui uma opção de compilação capaz de alterar a plataforma para a qual o programa é compilado (CPU ou GPU): THRUST_DEVICE_SYSTEM. Além disso, Thrust possui diferentes *namespaces* que podem ser usados no programa para definir o local de execução de um algoritmo (exemplos: **thrust::omp::vector** ou **thrust::cuda::vector**).

O exemplo da Figura 7.41 ilustra brevemente o uso de Thrust que ordena 32M números aleatórios em paralelo numa GPU (Thrust, 2012).

```
01  #include <thrust/host_vector.h>
02  #include <thrust/device_vector.h>
03  #include <thrust/generate.h>
04  #include <thrust/sort.h>
05  #include <thrust/copy.h>
06  #include <algorithm>
07  #include <cstdlib>
08
09  int main(void) {
10    // gera 32M números aleatórios sequencialmente
11    thrust::host_vector<int> h_vec(32 << 20);
12    std::generate(h_vec.begin(), h_vec.end(), rand);
13
14    // transfere os dados para a GPU
15    thrust::device_vector<int> d_vec = h_vec;
16
17    // ordena os dados na GPU (846M chaves/s numa GeForce GTX 480)
18    thrust::sort(d_vec.begin(), d_vec.end());
19
20    // transfere os dados de volta ao host
21    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
22
23    return 0;
24  }
```

Figura 7.41: exemplo da biblioteca Thrust (Fonte: Thrust, 2012)

A Figura 7.41 mostra a utilização de Thrust de forma muito semelhante ao uso da STL, assim como a programação heterogênea combinando a execução de algoritmos na CPU e na GPU. Na linha 11 é inicializado um **vector** no *host* (CPU) e na linha 12 ele é preenchido com números aleatórios. Na linha 15 uma simples atribuição é utilizada para fazer a transferência do **vector** do *host* para o dispositivo (GPU). Na linha 18 o algoritmo de ordenação (**sort**) é executado na GPU e na linha 21 ele é copiado para o vector original localizado no *host*.

7.10. Conclusão

Este tutorial apresentou o que tem sido chamado de computação heterogênea a partir dos ambientes e ferramentas mais utilizados para a computação combinada em CPU e GPU. Tais dispositivos tem sido referenciados como aceleradores e se juntam a outras opções tais como os FPGAs mencionados no início do tutorial.

Sem dúvida, as GPUs são os aceleradores em evidência no momento e possuem uma série de opções de ambientes e ferramentas de programação. Este tutorial apresentou algumas das arquiteturas de GPUs mais importantes, tais como a Fermi e a Kepler, assim como abordou ambientes de programação já tradicionais, tais como OpenMP, CUDA e OpenCL, além de abordagens mais modernas como OpenACC e Thrust.

Entretanto, esta é uma área em rápida e constante evolução. Novas arquiteturas e ambientes de programação são esperados para os próximos anos e este tutorial não encerra todas estas novas possibilidades. Entretanto, existe uma forte tendência que indica que a programação heterogênea recairá em modelos de paralelismo do tipo SIMD ou SIMT, que foram amplamente cobertos neste tutorial.

Bibliografia

Barney, B. (2012) OpenMP, Lawrence Livermore National Laboratory, UCRL-MI-133316 (disponível em https://computing.llnl.gov/tutorials/openMP/ , acessado em maio de 2012).

Borkar, S.; Chien, A. A. 2011. The future of microprocessors. Commun. ACM 54, 5 (May 2011), 67-77. DOI=10.1145/1941487.1941507. (disponível em: http://doi.acm.org/10.1145/ 1941487.1941507, acessado em maio de 2012).

Boyd, C., Huang, X., Pritchard, C., Kev Gee (2010). DirectCompute: A Teraflop for Everyone. GameFest 2010.

Bridges, T. (1990). The gpa machine: a generally partitionable msimd architecture. In Frontiers of Massively Parallel Computation, 1990. Proceedings., 3rd Symposium on the, pages 196–203.

Brodtkorb, A. R., Dyken, C., Hagen, T. R., & Hjelmervik, J. M. (2010). State-of-the-art in heterogeneous computing. Scientific Programming, 18, 1-33. doi:10.3233/SPR-2009-0296

Chapman, B.; Jost, G.; Pas, R. Using OpenMP: Portable Shared Memory Parallel Programming (2008). Scientific and Engineering Computation. [S.l.]: The MIT Press, 2008. ISBN 0262533022, 9780262533027.

Crawford, C. H., Henning, P., Kistler, M., & Wright, C. (2008). Accelerating computing with the cell broadband engine processor. Proceedings of the 5th conference on Computing frontiers, 3. ACM. Retrieved from http://portal.acm.org/citation.cfm?doid=1366230. 1366234

Darema-Rogers, D. A. G. V. A. N. and Pfister, G. F. (1985). A vm parallel environment. Technical Report RC-11225 (#9161), IBM Thomas J. Watson Research Center.

Darema, F. (2001). The spmd model: Past, present and future. In Proceedings of the 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, page 1, London, UK. Springer-Verlag.

Dongarra, J.; Lastovetsky, A. High Performance Heterogeneous Computing. Wiley-Interscience, 1 edition, 2009.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J. (2010). From CUDA to OpenCL : Towards a Performance-portable Solution for Multi-platform. Parallel Computing, (UT-CS-10-656), 1-12. Retrieved from http://www.sciencedirect.com/science /article/pii/S0167819111001335

Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. IEEE Trans. Comput., C-21:948+.

GCC (2011). Manual do GCC: The C Preprocessor. (disponível em: http://gcc.gnu.org /onlinedocs/cpp/index.html, acessado em maio de 2012).

JOCL (2012) Java Bindings for OpenCL. (disponível em: http://www.jocl.org/, acessado em maio de 2012).

JCUDA (2012) Java Bindings for CUDA. (disponível em: http://www.jcuda.org/, acessado em maio de 2012).

Kam, V. Introducing the DirectCompute Lecture Series!. DirectX Developer Blog. MSDN Blogs. 15 junho. 2010. (disponível em: http://blogs.msdn.com/b/directx/archive/2010/ 06/15/introducing-the-directcompute-lecture -series.aspx, acessado em maio de 2012).

Kirk, D. B. and meiW. Hwu,W. (2010). Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 1 edition.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A. (2012) PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, Parallel Computing, Volume 38, Issue 3, March 2012, Pages 157-174.

KRONOS Group. (2011). OpenCL - The open standard for parallel programming of heterogeneous systems. (disponível em: http://www.khronos.org/opencl/, acessado em maio de 2012).

Kunzman, D. M., & Kalé, L. V. (2009). Towards a framework for abstracting accelerators in parallel applications: experience with Cell. Cell, Dijon, Fra(c), 1-12. ACM. Retrieved from http://portal.acm.org/citation.cfm?id=1654114

MUNSHI, A., GASTER, B., MATTSON, T.G., FUNG, J., GISBURG, D. (2011) OpenCL Programming Guide. New York: Addison-Wesley Professional, 2011.

NVIDIA Corporation (2009). Fermi Compute Architecture White Paper. Version 1.1. (disponível em: http://www.nvidia.com/content/PDF/fermi_white_papers/ NVIDIA_Fermi_ Compute_Architecture_Whitepaper.pdf, acessado em maio de 2012).

NVIDIA Corporation (2012a). NVIDIA CUDA C Programming guide. Version 4.2. (disponível em: http://developer.download.nvidia.com/compute/DevZone/docs/ html/C/doc/ CUDA_C_Programming_Guide.pdf, acessado em março de 2012).

NVIDIA Corporation (2012b). NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110 White Paper, The Fastest, Most Efficient HPC Architecture Ever Built. Version 1.0. (disponível em: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, acessado em maio de 2012).

NVIDIA CUDA SDK (2012) Software Development Kit. Versão 4.2. (disponível em: http://developer.nvidia.com/cuda-downloads, acessado em maio de 2012).

OLCF, Oak Ridge Leadership Computing Facility Tutorial. 2012. (disponível em: http://www.olcf.ornl.gov/training_articles/opencl-vector-addition/, acessado em abril de 2012).

OpenACC (2011a) The OpenACC Application Programming Interface. Version 1.0. November, 2011. (disponível em: http://www.openacc.org/sites/default/files/ OpenACC.1.0_0.pdf, acessado em maio de 2012).

OpenACC Standard. (2011b). (disponível em: http://www.openacc-standard.org/, acessado em março de 2012).

OpenMP Site (2012). Site do OpenMP. (disponível em: http://openmp.org, acessado em maio de 2012).

PGI OpenACC 2012 Getting Started Guide. The Portland Group, Inc. and STMicroelectronics, Inc. Printed in the United States of America. Release 2012, version 12.4, April 2012. (disponível em: http://www.pgroup.com/doc/openACC_gs.pdf, acessado em maio de 2012).

Silva, L. e Stringhini, D. Paralelização de Aplicações para Realidade Aumentada em CUDA e OpenCL. Minicurso apresentado no *XIV Symposium on Virtual and Augmented Reality*, 28-31 de maio de 2012, Niterói/RJ, Brasil.

Storaasli, O. O., & Strenski, D. (2007). Exploring Accelerating Science Applications with FPGAs. Computing, July. (disponível em: http://ft.ornl.gov/~olaf/www/pubs/OlafRSSI2July 07.pdf, acessado em maio de 2012).

Thrust (2012). Thrust. (disponível em: http://thrust.github.com/, acessado em maio de 2012).

Wolfe, M. (2012) The Heterogeneous Computing Jungle. HPC Wire, Março, 2012. (disponível em: http://www.hpcwire.com/hpcwire/2012-03-19/the_heterogeneous_program ming_jungle.html, acessado em abril, 2012).