Programação Paralela - OPRP001
Unidades de Processamento Gráfico e CUDA

Lucas Leando Nesi, Guilherme Piegas Koslovski, Mauricio Aronne Pillon

[1]Departamento de Ciência de Computação (DCC)
Programa de Pós Graduação em Computação Aplicada (PPGCA)
Universidade do Estado de Santa Catarina - UDESC - Joinville, SC - Brasil

lucas.nesi@edu.udesc.br, {guilherme.koslovski, mauricio.pillon}@udesc.br

1. Unidade de Processamento Gráfico

As Unidades de Processamento Gráfico (GPU, do inglês *Graphical Processing Unit*) são dispositivos que foram criados originalmente para o processamento em tempo real de primitivas gráficas. O poder de processamento de um único dispositivo pode chegar aos TeraFLOPS (*Floating Point Operations Per Second*). No campo da computação gráfica, as GPUs são responsáveis por realizar operações geométricas paralelamente para a rasterização de ambientes virtuais. Porém, sua utilização está sendo expandida para uma vasta gama de aplicações devido ao seu poder de processamento e de banda de memória interna [Plauth et al. 2016, Govindaraju et al. 2006].

A capacidade da arquitetura SIMD (*Single Instruction Multiple Data*) presente nas GPUs permite a utilização de centenas de unidades paralelas de ponto flutuante, presentes em um único dispositivo, que pode ser adquirido por poucas centenas de reais [Hennessy and Patterson 2011]. Uma das classes da taxonomia de [Flynn 1972], as arquiteturas SIMD utilizam o paralelismo em nível de dados como base de sua execução. De forma geral, várias unidades de processamento inteiro ou de ponto flutuante executam as mesmas instruções para diferentes fluxos de dados. Com SIMD é necessário apenas uma única memória de instruções e um único processador de controle para várias unidades de processamento, cada qual com sua memória de dados [Tanenbaum 2007]. Desta forma, esta arquitetura possibilita maiores quantidades de unidades de processamento por dispositivo e a diminuição de redundância no controle de fluxo das mesmas instruções. Estas instruções seriam executadas várias vezes serialmente de forma idêntica em diferentes conjuntos de dados nas arquiteturas SISD (*Single Instruction Single Data*) ou MIMD (*Multiple Instruction Multiple Data*) [Hennessy and Patterson 2011].

De modo geral, a arquitetura de uma GPU é composta por vários níveis de memória e processadores integrados no próprio dispositivo conectados com o resto dos componentes do computador via interface PCI-Express [Cook 2013]. Todo o controle de execução e transferência de memória se dá por interfaces de programação, como CUDA (*Compute Unified Device Architecture*), responsáveis por transferir os dados necessários entre a memória RAM e a memória interna dos dispositivos, assim como controle sobre a execução de programas SIMD nas GPUs, chamados de *kernels* [NVIDIA 017a].

Em 2007 a NVIDIA criou o modelo de programação CUDA, utilizando a linguagem C/C++ para auxiliar a programação genérica de suas GPUs [NVIDIA 017a]. Um modelo de programação similar é o OpenCL, desenvolvido por empresas e pela comunidade para oferecer uma interface independente de fornecedor para múltiplas plataformas e dispositivos [Hennessy and Patterson 2011]. Pela melhor consolidação do modelo

CUDA, e pela disponibilidade de dispositivos da NVIDIA, o presente trabalho usará o modelo CUDA e as unidades de processamento gráfico da NVIDIA.

2. Modelo de Programação CUDA

No modelo de programação CUDA, os programadores criam programas heterogêneos usando a linguagem de programação C/C++ para CPU, que é chamada de *host*. Estes programas podem chamar funções SIMD, apelidadas de *kernels*, na GPU, que é chamada de *Device*. Os *kernels* ainda podem chamar funções extras dentro da GPU mas não podem invocar funções no *host*. Todas as funções que devem ser executadas e invocadas em CPU devem ser declaradas como funções *__host__* e todos os *kernels*, que serão executados em GPU mas serão invocados em CPU devem ser declarados como funções *__global__*. Ainda, as funções que devem ser executadas e invocadas em GPU devem ser declaradas como funções *__device__* [NVIDIA 017a]. Então, quando um programa CUDA é compilado, as funções que devem ser executadas em CPU/*host* serão compiladas normalmente, enquanto os *kernels* e as funções *Device* serão compilados para a arquitetura da GPU. Quando uma função *kernel* é invocada em CPU, é função do sistema operacional, mais precisamente dos *drivers*, módulos de conexão com a GPU e da API (*Application Programming Interface*) CUDA, realizar a transferência da função *kernel* para a GPU e requisitar a execução da mesma [Cook 2013].

A menor unidade paralela em CUDA é a *thread*, que é usada como primitiva de programação e onde são executadas instruções para apenas um fluxo de dados. Um agrupamento de *threads* é chamada de bloco, e cada bloco será alocado para uma unidade arquitetural chamada de *Streaming Multiprocessor* (SM). Um conjunto de blocos é chamado de grade de *threads*. As *threads* são executadas em grupos de 32 *threads*, chamado de *warp*.

Para a chamada de uma função *kernel* é necessário informar o tamanho da grade de *threads*, a quantidade de blocos totais e a quantidade de *threads* por bloco. Estes valores são definidos pelo programador com base no problema em questão e respeitando as limitações do dispositivo [NVIDIA 017a]. Por exemplo, para a realização da soma de dois vetores, cada um contendo 512 elementos, pode-se chamar o *kernel* Soma apresentado no Algoritmo 1 com apenas 1 bloco com 512 *threads*. Cada *thread* será responsável pela soma de 1 elemento de cada vetor, e como existe apenas 1 bloco, todas as 512 *threads* estarão necessariamente em um *Streaming Multiprocessor* (SM). Porém, o mesmo *kernel* poderia ser invocado utilizando 2 blocos com 256 *threads* cada um, novamente cada *thread* será responsável pela soma de 1 elemento em cada vetor, mas cada bloco poderá ser alocada em diferentes SMs. A decisão da quantidade de blocos a serem abertos depende de algumas informações de *Hardware*.

Algoritmo 1: *kernel* de soma de vetores

```
1 __global__ void Soma(int* A, int* B, int* C)
2 {
3     unsigned int id = threadIdx.x + blockIdx.x * blockDim.x;
4     C[id] = A[id] + B[id];
5 }
```

A Figura 1 mostra a hierarquia entre grade, bloco e *threads*. O tamanho de qual-

quer um destes elementos pode ser um vetor tridimensional, ou seja, pode-se criar um *kernel* com grade de tamanho (3,3,3) blocos, sendo um total de 27 blocos. Na programação, cada bloco, *thread* e *warp* recebem um identificador único, acessado usando variáveis pré-definidas pelo modelo CUDA. No caso das *threads*, o identificador é em relação ao seu bloco e esta variável é chamada *threadIdx*. Para os blocos, seu identificador é a variável *blockIdx*. É com base nesta identificação que é realizada a escolha e seleção do fluxo de dados que cada *thread* irá utilizar. Além da identificação de unidade, pode-se obter as dimensões de qualquer nível da grade. Para descobrir em tempo de execução qual é a dimensão de um bloco utiliza-se a variável *blockDim*, que informará quantas *threads* cada bloco possui. E no caso de quantos blocos por grade, é utilizada a variável *gradDim*. Como os elementos podem ser vetores tridimensionais, estas variáveis de identificação possuem sub elementos x, y, z que informam respectivamente seus tamanhos na primeira, segunda e terceira dimensão [NVIDIA 017a]. O *kernel* Soma apresentado no Algoritmo 1 mostra como podem ser utilizadas estas variáveis. Como apenas estruturas de uma dimensão são utilizadas (vetores), o *kernel* foi invocado utilizando apenas uma dimensão, e portanto o acesso as variáveis de identificação é feito utilizando somente o sub elemento x.

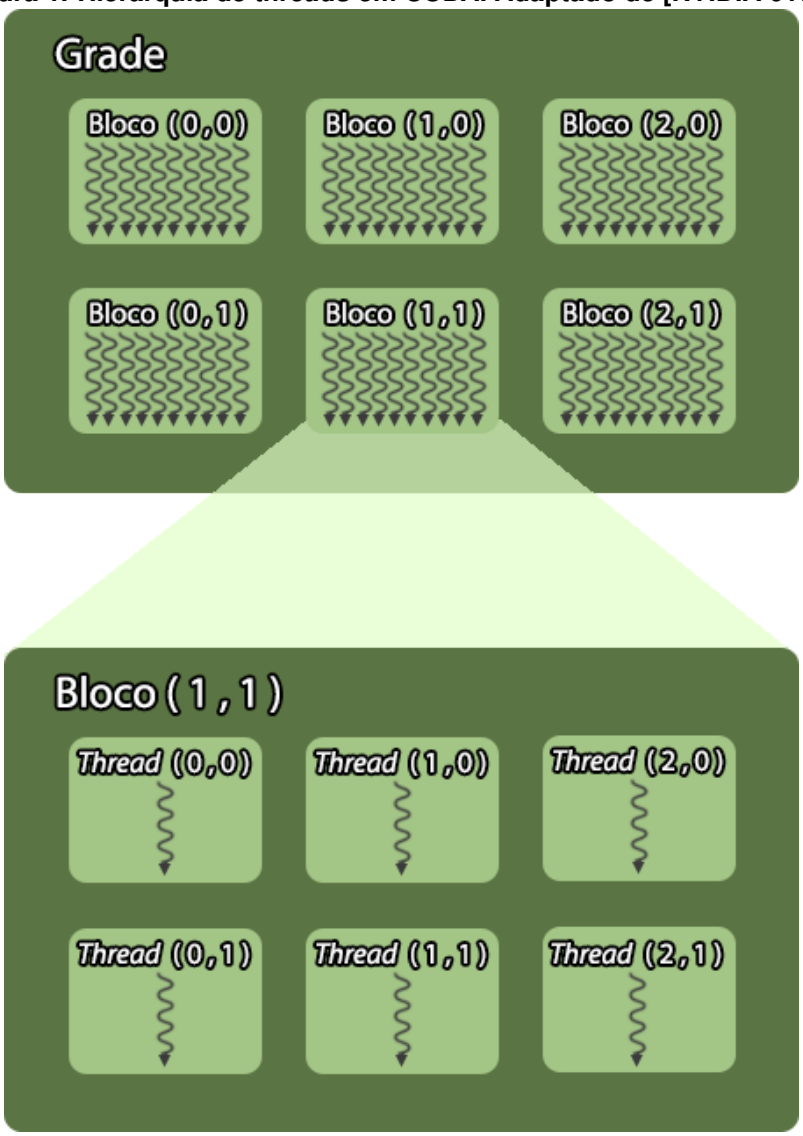
A memória interna da GPU é independente da memória RAM. Desta forma, um programa em CPU não pode acessar a memória global da GPU e vice-versa. Para realizar esta comunicação, são necessárias chamadas da API CUDA para a alocação e transferência da memória entre os dispositivos. Na sua grande maioria, estas chamadas devem ser realizadas em funções em *host* e devem ser invocadas antes de utilizar dados que foram modificados ou criados pela outra parte. A NVIDIA, a partir da versão do modelo CUDA 6.0, introduziu o acesso unificado de memória, no qual o programador pode utilizar os mesmos ponteiros de memória tanto em *host* como em *Device*. Isso é possível, já que inserções de chamadas específicas da API CUDA para a alocação dos dados e de transferência de memória entre os dispositivos é realizada em tempo de compilação [NVIDIA 017a].

Por fim, existem três escopos de memória internas da GPU em CUDA: (i) Memória global, que pode ser acessada por todas as *threads* de qualquer bloco. Além disso, as transferência da memória RAM são feitas para a memória global. (ii) Memória compartilhada, disponível internamente para cada bloco. Apenas as *threads* do mesmo bloco podem acessar a mesma memória compartilhada e a modificação da memória compartilhada por uma *thread* é vista por qualquer outra *thread* do mesmo bloco. A velocidade de acesso da memória compartilhada é superior ao da memória global. (iii) Memória local de *thread*, que são basicamente todas as variáveis locais de *threads* [Cook 2013]. A Figura 2 mostra os diferentes escopos de memória internas da GPU disponíveis para cada elemento da grade.

3. Arquitetura de GPUs NVIDIA

As GPUs da NVIDIA são divididas em três grupos comerciais: (i) Geforce, voltado para o entretenimento, especialmente para a execução de jogos de computador. (ii) Quadro, dedicado para os profissionais das áreas gráficas, para a criação de componentes gráficos como modelos n-dimensionais, animações e simulações. (iii) Tesla, dedicado para a computação de alto desempenho, especialmente para o processamento científico e comercial de dados em larga escala.

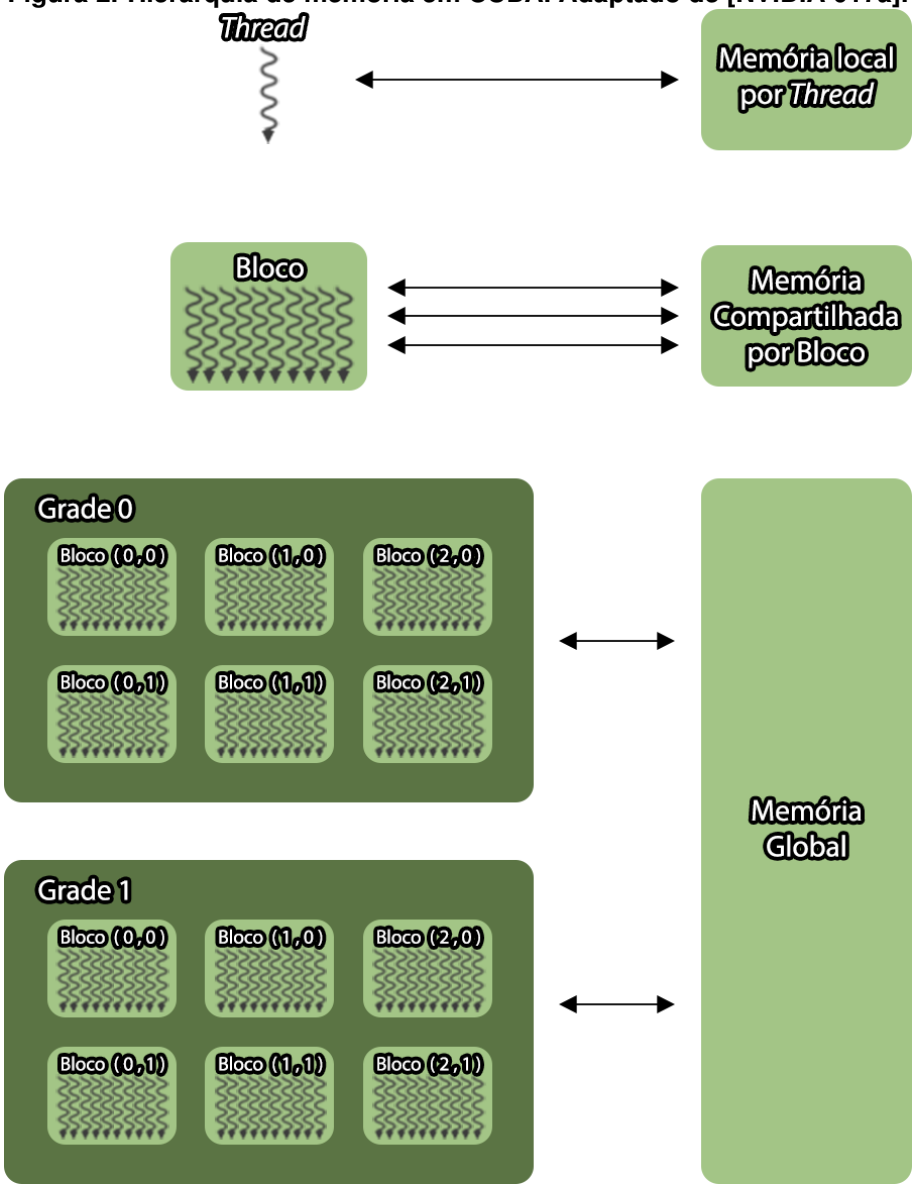
Figura 1. Hierarquia de *threads* em CUDA. Adaptado de [NVIDIA 017a].



A NVIDIA separa as gerações de suas placas por um número de versão chamado capacidade de computação (*Computer Capability*). Esta versão de arquitetura é independente da versão do modelo CUDA. Cada grande versão de capacidade de computação possui um nome, por exemplo, a versão 6.x é apelidada de *Pascal*, enquanto a 3.x é chamada de *Kepler*. Para cada versão de capacidade de computação existe a adição de novas funções e diferenças nas arquiteturas. Por exemplo, a capacidade de uma GPU da NVIDIA de processar ponto flutuante de meia-precisão só está disponível a partir da versão 5.3, e a utilização de memória unificada apenas com versões superiores a capacidade de computação 3.0 [NVIDIA 017a, Cook 2013]. Uma listagem de algumas placas e suas respectivas versões de capacidade de computação é mostrada na Tabela 1. Os dispositivos disponíveis para realização do presente trabalho são a GeForce GTX 730 128 bits e a GeForce GTX 1080.

Uma GPU é composta por três componentes principais: (i) *Streaming Processor* (SP), ou Cuda Core, responsável pelo processamento de uma *thread*. (ii) *Streaming Multi-*

Figura 2. Hierarquia de memória em CUDA. Adaptado de [NVIDIA 017a].



processor (SM), responsável pelo processamento total de um ou mais blocos que contêm vários Cuda Cores. (iii) Memórias, classificadas e localizadas em diferentes partes da GPU. Especificamente, as memórias podem ser classificadas como:

- (i) Global, uma por dispositivo, recebe os dados da memória RAM e pode ser vista por qualquer *thread* de qualquer bloco de qualquer grade.
- (ii) Cache L2 para memória global, uma por dispositivo. Nas últimas versões de capacidade de computação possui linhas de 128B.
- (iii) Constante, só pode ser acessada e é alocada em memória global.
- (iv) Compartilhada, uma por *Streaming Multiprocessor* (SM), que são divididas em bancos menores. Para realizar o uso de memória compartilhada o programador deve fazer sua declaração explícita.
- (v) Cache L1/Textura, internas do *Streaming Multiprocessor* (SM), são usadas para as trocas entre registradores, informações sobre a pilha de execução ou arma-

Tabela 1. Versão de capacidade de computação para algumas GPUs NVIDIA. Adaptado de [NVIDIA 017a].

Capacidade de Computação	Nome Arquitetura	GPUs	GeForce	Tesla
1.0	Tesla	G80	8800, 8800 GTX	C870, D870, S870
1.1		G92, G94, G96, G98, G84, G86	GTS 250	-
1.2		GT218, GT215	GT 340	-
1.3		GT200, GT200b	-	C1060, S1070
2.0	Fermi	GF100, GF110	GTX 590	C2075, C2070, M2050, M2070, M2075, M2090
2.1		GF104, GF108, GF114, GF119	GTX 730 128 bits	-
3.0	Kepler	GK104, GK107	GTX 730 256 bits	K10, K340, K520
3.5		GK110, GK208	GTX 780, GT 740	K40, K20x, K20
3.7		GK210	-	K80
5.0	Maxwell	GM107, GM108	-	M10
5.2		GM200, GM206	GTX 980, GTX 970	M4, M40, M6, M60
5.3		GM20B	-	-
6.0	Pascal	GP100	-	P100
6.1		GP102, GP104, GP106, GP107	Titan X, GTX 1080 GTX 1070, GTX 1050	P40, P4
7.0	Volta	GV100	-	V100

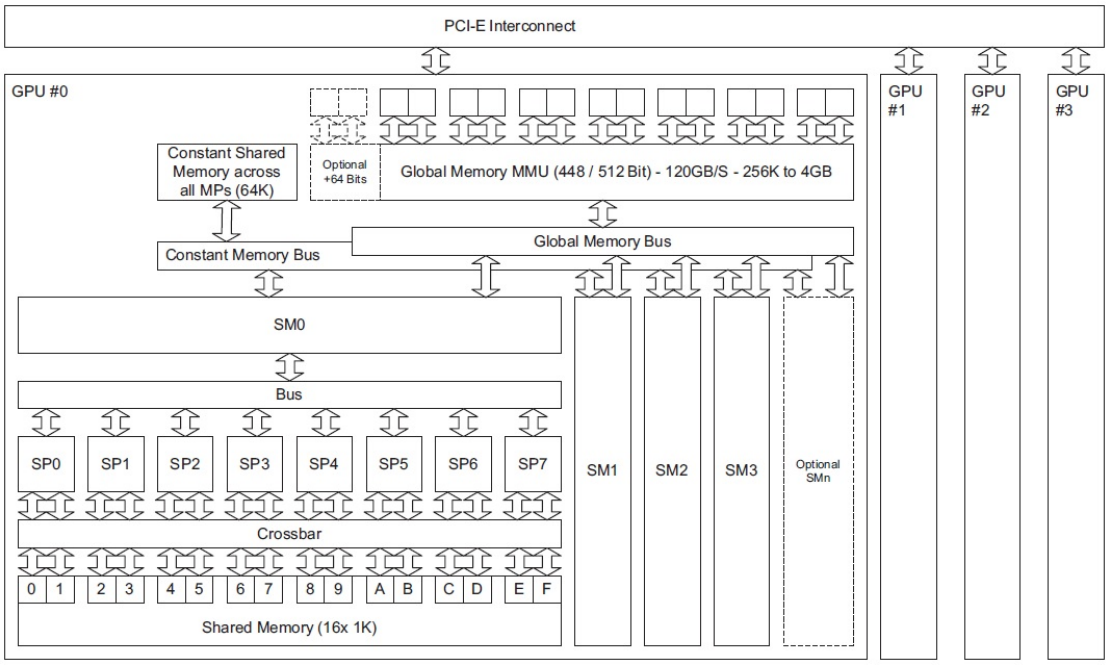
zenamento de texturas. Pode ser ativada também como Cache global.

- (vi) Registradores, internas do *Streaming Multiprocessor* (SM), onde existe um banco de registradores. No caso da versão de capacidade de computação superior a 5.0 existem 64K de registradores de 4 Bytes por SM. Todas as variáveis locais de cada *thread* serão alocadas em registradores. Além disso, é necessário um registrador para a identificação da *thread* [Hennessy and Patterson 2011].

Uma GPU contém vários *Streaming Multiprocessors* (SM) que por sua vez contêm vários Cuda Cores. A quantidade de *Streaming Multiprocessors* (SM) por dispositivo e de Cuda Cores por SM varia para cada capacidade de computação. A comunicação entre a GPU e a CPU é feita pela *Gigathread Engine* [NVIDIA 016a]. A Figura 3 mostra o diagrama simplificado da GPU NVIDIA (G80/GT200), onde a GPU 0 está conectada com o resto do equipamento com o barramento PCI-Express, que pode ter outras GPU conectadas. Todos os SMs estão conectados em um barramento único de acesso a memória global e constante. Os SPs de uma SM também estão conectados com a memória compartilhada em seus 16 bancos (versão de capacidade de computação 1.0).

As principais diferenças entre as arquiteturas de GPUs são as quantidades de SMs e de Cuda Cores por SM. Nota-se também que as GPUs após a versão de capacidade de computação 3.0 possuem barramento PCI express 3.0 (x16 = 15.754 GB/s) e que as SMs estão agrupadas em GPC (*Graphics Processing Clusters*). Cada GPC contém um Raster Engine que é utilizado nas *pipelines* gráficas, como por exemplo, OpenGL ou

Figura 3. Diagrama simplificado da GPU NVIDIA (G80/GT200) [Cook 2013].



DirectX [NVIDIA 016a].

Por exemplo, na Figura 4 existe o diagrama da SM da Geforce GTX 1080 utilizada no presente trabalho. Cada SM contém 128 Cuda Cores, agrupados em 32 Cuda Cores para cada *Instruction Buffer* e *Warp Scheduler*. Este agrupamento justamente acontece pela dimensão do *warp*, o *Warp Scheduler* é o escalonador de instruções SIMD, também conhecida como instruções PTX (*Parallel thread Execution*), para aquele grupo de 32 Cuda Cores. Sua função é decidir qual das instruções PTX disponíveis no *Instruction Buffer* está pronta para execução e enviar para o *Dispatch Unit*, que por sua vez, envia para os Cuda Cores ou unidades especiais.

Ainda, cada SM da Geforce GTX 1080 também contém 96KB de memória compartilhada e Caches L1/Textura. E cada agrupamento contém bancos de registradores de 4 Bytes totalizando 65536 registradores por SM, 8 unidades de SFU (*Special Function Unit*) capazes de executar instruções especiais em *hardware* e 8 *LD(Load)/ST(Store) Units*, unidades de busca e gravação de memória [NVIDIA 016a, Cook 2013].

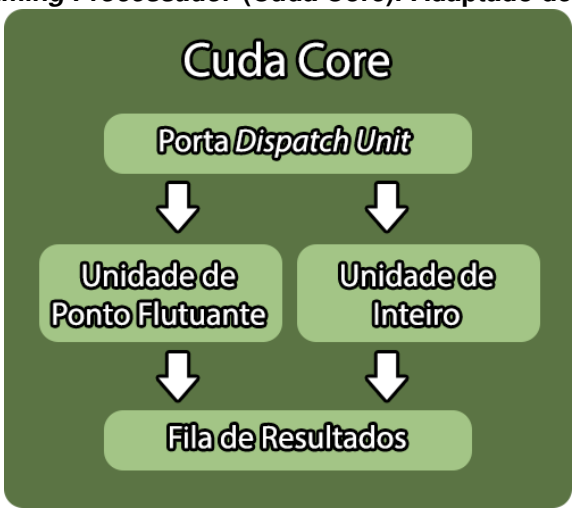
Um Cuda Core pode ser visto na Figura 5. Cada Cuda Core tem uma unidade de processamento de ponto flutuante e de inteiro [Hennessy and Patterson 2011]. Existem Cuda Cores com processamento de dupla precisão que estão presentes em arquiteturas de alto desempenho (Tesla), como é no caso da arquitetura Pascal GP10, onde cada SM contém 64 Cuda Cores de precisão simples e 32 Cuda Cores de dupla precisão [NVIDIA 016b]. Apesar de algumas arquiteturas não apresentarem Cuda Cores de dupla precisão, o suporte para o processamento de dupla precisão é emulado/processado pelos Cuda Cores de precisão simples utilizando mais ciclos que a mesma operação com ponto flutuante simples [NVIDIA 016a].

Assim, para um *kernel* ser executado, uma grade de *threads* deve ser informada e cada grade é alocada para uma GPU. A Gigathread Engine é responsável pela

Figura 4. *Streaming Multiprocessor* (SM) da GeForce GTX 1080 [NVIDIA 016a].

comunicação entre GPU e CPU, decidindo para qual *Streaming Multiprocessor* (SM) cada bloco deve ser alocado. Quando um bloco é alocado em um SM, ele é dividido em vários *warps* e cada um é enviado para um *Warp Scheduler* que decidirá quando este será execu-

Figura 5. *Streaming Processador* (Cuda Core). Adaptado de [NVIDIA 017a].



tado nos seus Cuda Cores. Por último, cada uma das 32 *threads* de um *warp* será alocada para 32 Cuda Cores diferentes. Cada instrução PTX será realizada individualmente em uma *thread*. Em cada *warp* sempre 32 *threads* estarão executando a mesma instrução PTX, isto é chamado de paralelismo de *warp*, utilizado para controle de sincronia entre *threads*. Porém *threads* de um mesmo bloco, alocados em diferentes *warps* podem ser executadas em tempos diferentes. Quando é necessário que todas as *threads* de diferentes *warps* cheguem a certa parte do código, é usado uma sincronia em barreira [Cook 2013].

4. Limitadores de Desempenho em GPUs

Nem todos os algoritmos são adequados para a utilização de arquiteturas SIMD. Isso pode ser resultante da baixa quantidade de partes que podem ser paralelizáveis ou pela técnica algorítmica utilizada não ser ajustada perfeitamente aos requisitos das GPUs. Nesta subseção, são discutidos algumas situações que podem prejudicar o desempenho dos programas na GPU e métricas utilizadas para aferir a eficiência da execução [NVIDIA 017b].

A primeira métrica a ser discutida é a Ocupação de GPU, que significa a quantidade de recursos que estão sendo utilizadas em uma SM. Ela é calculada pela quantidade de *warps* escalonados para uma SM, dividido pela quantidade máxima possível de *warps* residentes por SM. A partir da versão de capacidade de computação 3.0, a quantidade máxima de *warps* residentes por SM é de 64, ou seja, para ser atingido a ocupação de 100% devem ser escalonados 64 *warps* para uma SM, podendo ser de diferentes blocos. A ocupação é uma das principais métricas em GPU, pois avalia o tempo ocioso dos Cuda Cores. Como as GPUs fazem o escalonamento entre *warps* rapidamente, a quantidade extra de *warps* residentes é usada para maximizar a utilização das unidades lógicas de inteiro e de ponto flutuante. Então, para cada operação que deixe uma das *threads* de um *warp* em estado de espera, como acesso a memória, outro *warp* é escalonado para a execução naqueles Cuda Cores [NVIDIA 017b].

Existem alguns fatores que podem limitar a ocupação da GPU, como por exemplo, quando a quantidade de blocos alocados na SM é o máximo possível, porém eles não têm a quantidade de *threads* suficientes para maximizar o uso da SM. A utilização de registradores e memória compartilhada limita o número de blocos por SM. Não podem ser

alocados simultaneamente para uma SM, blocos que tenham seus recursos somados necessários ultrapassando os limites físicos de memória compartilhada e número de registradores. Apesar de individualmente serem válidos, estes blocos devem ser escalonados para diferentes SMs. Ainda, mesmo que um bloco utilize o limite máximo de *threads*, 1024, a SM não atingirá a ocupação máxima, porque serão criados apenas 32 *warps*, causando uma ocupação teórica máxima de 50%. Deve-se sempre limitar e calcular a quantidade de registradores e memória compartilhada sendo usada para que o maior número possível de blocos possa ser escalonado para uma SM [NVIDIA 017b].

Destaca-se também outro limitante de desempenho nas GPUs que é o padrão de acesso às memórias. O acesso da memória global é intermediado pela cache L2. Algumas práticas de acesso de memória garantem o melhor uso possível da cache e por consequência a diminuição da utilização de banda entre L2 e a memória global. As linhas em L2 têm tamanho de 128 Bytes e quando uma variável é acessada em memória global ela e todos os outros 124 Bytes da mesma linha são transferidos para L2. Desta forma, se um *warp* faz uso de acessos lineares e respeitando as linhas de L2, apenas um acesso é realizado em memória global e as 32 *threads* conseguem seus dados diretos de L2 [Hennessy and Patterson 2011, NVIDIA 017b, Wang et al. 2016].

Como as GPUs utilizam arquitetura SIMD, várias instruções idênticas então sendo executadas em diferentes fluxos de dados. Porém, existe uma violação deste princípio quando há uma divergência de instruções no algoritmo dentro de um *warp*. Neste cenário, nem todas as *threads* estarão executando os mesmo blocos de comandos, como ocorre no caso de um desvio condicional (*if*). Para garantir o funcionamento do algoritmo, o controlador de instruções decidirá qual bloco executará primeiro, e todas as *threads* que não fazem parte dele executarão um comando nulo. O controlador de instruções realizará este mapeamento até que todas as *threads* confluam novamente para a mesma instrução. Este problema então, causa a diminuição de paralelismo, já que duas ou mais instruções diferentes não podem ser executadas ao mesmo tempo [Hennessy and Patterson 2011]. O escalonador ainda consegue otimizar divergências de apenas uma instrução em grupos de *half warp*, 16 *threads*, isso ocorre já que existe dois *Dispatch Units* por *Warp Scheduler*. Graças a isso, divergências que afetam precisamente metade do *warp* não sofrem problemas de perda de paralelismo [Cook 2013].

5. Conclusão

Uma das tecnologias de alto desempenho que está em destaque é a GPU, um dispositivo com arquitetura SIMD, altamente paralelizável, que tem um custo-benefício elevado. Cada GPU pode conter milhares de unidades de processamento e pode chegar aos TFLOPS de desempenho. Além disso, as GPUs são partes essenciais para a visualização gráfica, e portanto, presentes nos computadores pessoais. Todo este poder computacional está sendo usado em diversas áreas, principalmente após a consolidação dos modelos de programação para GPUs.

Para a programação de algoritmos genéricos em GPUs são utilizados modelos de programação como CUDA, que foi desenvolvido pela NVIDIA para seus dispositivos. Ele permite a criação de programas heterogêneos entre a CPU e a GPU. Algumas particularidades deste modelo são a invocação de funções na GPU que devem ser dimensionadas pela grade CUDA, contendo o número de blocos e *threads*, e os escopos de memória para

cada grupo de processamento. Para maximizar o desempenho de uma GPU, os algoritmos devem respeitar os seus padrões arquiteturais, sendo que nem todos os algoritmos são apropriados para este formato. Desta forma, a programação de algoritmos em GPUs não é trivial e requer: (i) Um mapeamento adequado das partes paralelizáveis nas grades CUDA, (ii) Uma programação compatível com a arquitetura da GPU e suas limitações.

Referências

Cook, S. (2013). *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960.

Govindaraju, N. K., Larsen, S., Gray, J., and Manocha, D. (2006). A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA. ACM.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

NVIDIA (2016a). GeForce GTX 1080 Whitepaper. *NVIDIA Corporation, Technical Report*.

NVIDIA (2016b). Tesla P100 Whitepaper. *NVIDIA Corporation, Technical Report*.

NVIDIA (2017a). *CUDA C Programming Guide, NVIDIA Documentation*. NVIDIA Corporation, Santa Clara, CA, USA.

NVIDIA (2017b). *CUDA C Best Practices Guide, NVIDIA Documentation*. NVIDIA Corporation, Santa Clara, CA, USA.

Plauth, M., Feinbube, F., Schlegel, F., and Polze, A. (2016). A performance evaluation of dynamic parallelism for fine-grained, irregular workloads. *International Journal of Networking and Computing*, 6(2):212–229.

Tanenbaum, A. (2007). *Organização estruturada de computadores*. PRENTICE HALL BRASIL, Rio De Janeiro, RJ, Brasil, 5 edition.

Wang, L., Wang, Y., Yang, C., and Owens, J. D. (2016). A comparative study on exact triangle counting algorithms on the GPU. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP '16, pages 1–8, New York, NY, USA. ACM.