3
Computação de Propósito Geral em Unidades de Processamento Gráfico

As Unidades de Processamento Gráfico (GPUs) foram originalmente desenvolvidas para o processamento de gráficos e eram difíceis de programar. Contudo, o desenvolvimento de interfaces de programação e o suporte a linguagens como o C permitiram que as GPUs fossem utilizadas para Computação de Propósito Geral (GPGPU) (Harris, 2005). Atualmente as GPUs são processadores massivamente paralelos, compostos por um grande número de elementos de processamento num único chip, com alto desempenho em operações de ponto flutuante.

Neste Capítulo descreveremos inicialmente alguns conceitos fundamentais da programação paralela, passando em seguida para o modelo de programação paralela desenvolvido especificamente para placas gráficas da NVIDIA, denominada de CUDA (*Compute Unified Device Architecture*).

3.1.
Programação Paralela

A programação paralela consiste em subdividir um problema em partes independentes de maneira que seja possível utilizar múltiplos elementos de processamento para executá-las simultaneamente (Barney, 2009). O interesse pela computação paralela tem aumentado devido às restrições físicas que impedem o aumento crescente da freqüência.

A Lei de Moore é uma observação empírica de que a densidade de transistores duplica a cada 18 ou 24 meses (Moore, 1998). A dificuldade de aumentar a freqüência faz com que os transistores adicionais sejam usados para adicionar hardware extra para computação paralela.

O paralelismo pode ocorrer em diversos níveis. O Paralelismo em Nível de Bit significa aumentar o tamanho da palavra, reduzindo a quantidade de instruções que um processador deve executar para realizar uma operação em variáveis cujo

tamanho é maior do que o da palavra. Um processador de 32 bits teria que executar duas operações para realizar a soma de dois números inteiros com precisão de 64 bits, enquanto um processador de 64 bits poderia fazê-lo em uma única operação.

O Paralelismo em Nível de Instrução consiste em reordenar o fluxo de instruções e reagrupá-las, de modo que estes grupos possam ser executados em paralelo sem alterar o resultado do programa. O *pipeline* dos processadores modernos possui múltiplos estágios. Cada estágio corresponde a uma ação diferente que o processador executa em determinada instrução; um processador com um *pipeline* de N estágios pode ter até N diferentes instruções em diferentes estágios de execução.

O Paralelismo de Dados requer a distribuição dos dados por diferentes nós computacionais de maneira a viabilizar a execução de uma mesma instrução sobre diferentes conjuntos de dados simultaneamente (Hillis e Steele, 1986). O paralelismo de dados é inerente aos laços de repetição e só pode ser empregado se não houver a dependência de uma iteração do laço com a saída de uma ou mais iterações anteriores.

Diferentemente do Paralelismo de Dados, no Paralelismo de Tarefas operações diferentes podem ser executadas sobre um mesmo conjunto de dados ou sobre conjuntos de dado diferentes.

3.1.1. Taxonomia de Flynn

A Taxonomia de Flynn é um modelo de classificação de arquitetura de computadores baseado no fluxo de instruções e dados (Flynn, 1972; Duncan, 1990). Esta classificação é dividida em quatro categorias: SISD, SIMD, MISD e MIMD.

A classificação SISD (*Single Instruction, Single Data*) equivale a um programa puramente seqüencial, sendo também conhecida como arquitetura Von Neumann. Um fluxo único de instruções é aplicado sobre um conjunto único de dados conforme o ilustrado na Figura 2. Na ilustração UC é utilizado para representar a unidade de controle e UP é usado para representar a unidade de processamento.

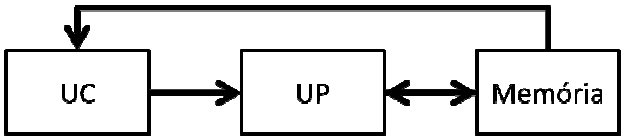

Figura 2 – Arquitetura SISD.

O modelo das arquiteturas vetoriais, onde uma operação é executada sobre múltiplos operandos simultaneamente, é denominada SIMD (*Single Instruction, Multiple Data*). A Figura 3 ilustra esta arquitetura.

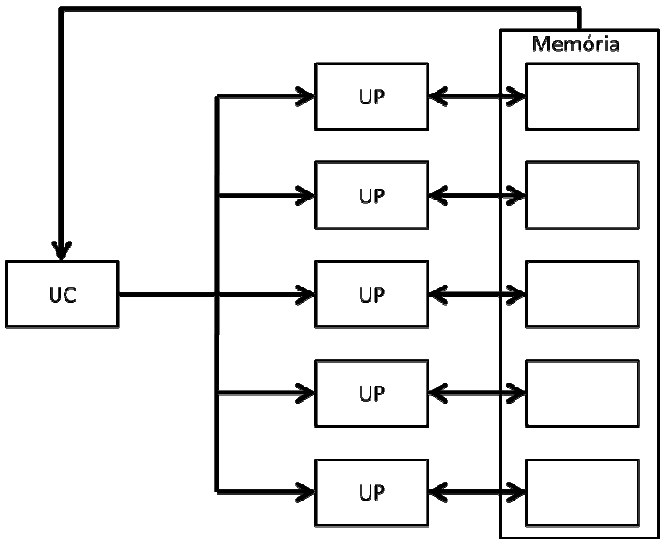

Figura 3 – Arquitetura SIMD.

Na classificação MISD (*Multiple Instruction, Single Data*), múltiplas unidades de processamento realizam operações diferentes sobre um mesmo conjunto de dados. Conforme o ilustrado na Figura 4, numa máquina de fluxo de dados, uma unidade de processamento realiza uma operação sobre o dado e o passa à unidade de processamento seguinte que executará uma operação diferente sobre o mesmo dado.

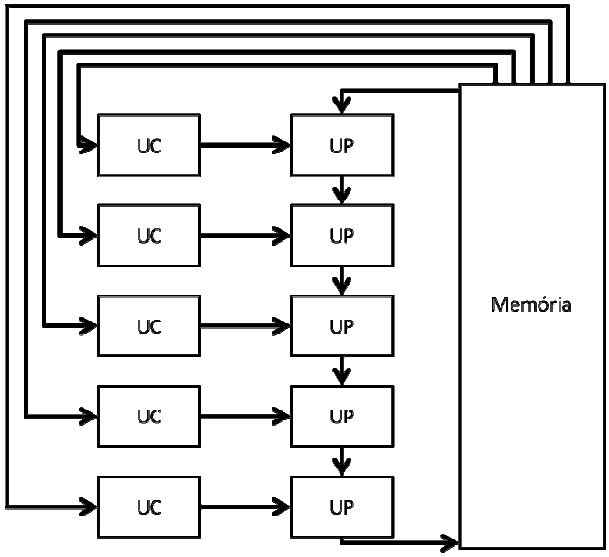

Figura 4 – Arquitetura MISD.

A arquitetura MIMD (*Multiple Instruction, Multiple Data*) empregada em multiprocessadores é apresentada na Figura 5. Nesta arquitetura, diferentes instruções são executadas sobre diferentes conjuntos de dados simultaneamente, usando unidades de processamento diferentes controladas por unidades de controle independentes.

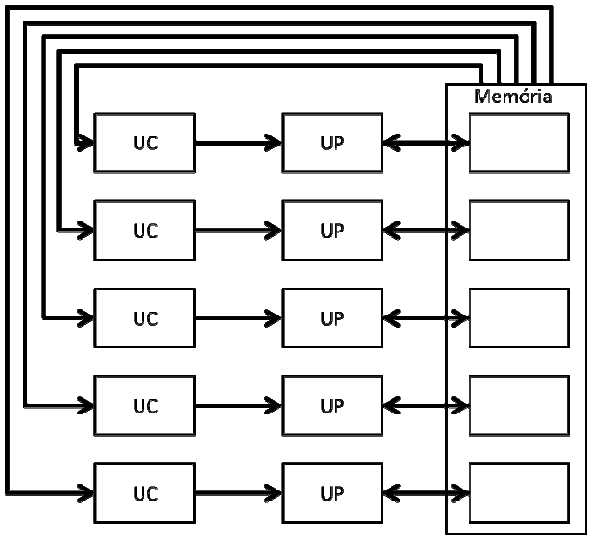

Figura 5 – Arquitetura MIMD.

3.1.2. Lei de Amdahl

O ganho de velocidade de um algoritmo rodando em múltiplos processadores é limitado pelo tempo consumido pela fração seqüencial do

programa. Assim, o máximo ganho de velocidade esperado quando um sistema não pode ser totalmente paralelizado é dado pela Lei de Amdahl (1967):

$$S = \frac{1}{1 - P + \frac{P}{N}} \tag{3.1}$$

Onde S é o ganho de velocidade do programa, P é a fração do programa que pode ser paralelizada e N é o número de processadores.

A Figura 6 ilustra o efeito da Lei de Amdahl para os casos em que uma fração de 50%, 75%, 90% ou 95% do programa podem ser efetivamente paralelizados.

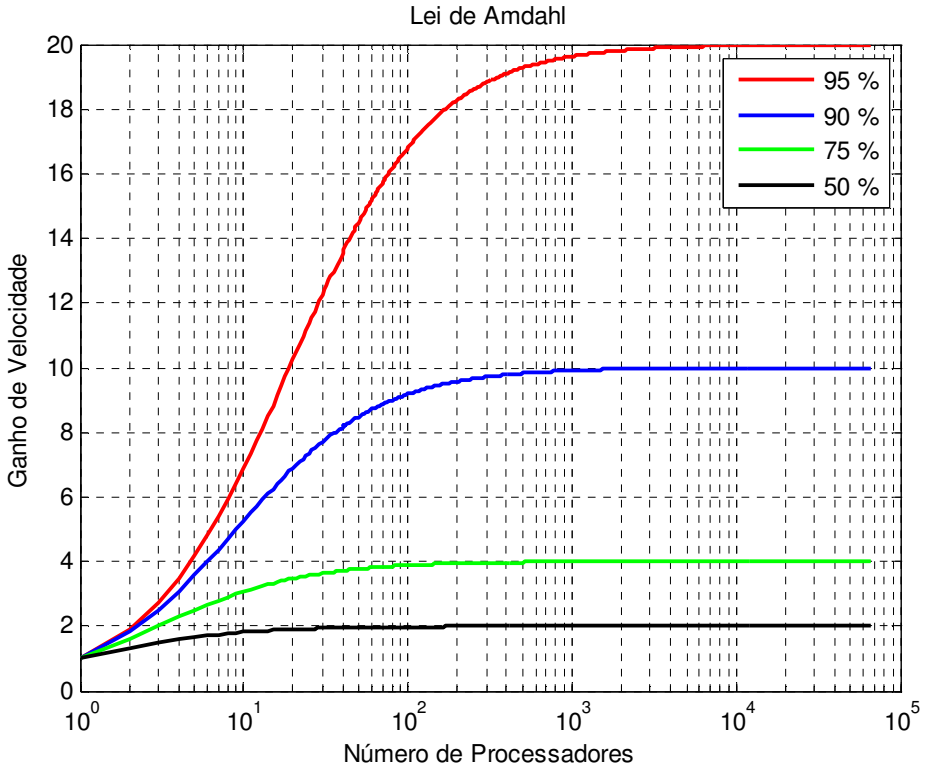

Figura 6 – Efeito da Lei de Amdahl sobre o ganho de velocidade de algoritmo rodando em múltiplos processadores.

3.2. Compute Unified Device Architecture (CUDA)

Em Novembro de 2006, a NVIDIA apresentou uma extensão da linguagem C para programação em GPUs, denominada CUDA (*Compute Unified Device Architecture*). Esta linguagem simplificada possibilitou a utilização da alta capacidade de cálculos das GPUs para solucionar problemas de propósito geral no meio científico (NVIDIA, 2010).

A demanda, no mercado de jogos, por gráficos 3D de alta definição em tempo real estimulou o desenvolvimento das GPUs. As GPUs se tornaram processadores de múltiplos elementos paralelos (NVIDIA, 2009b). A Figura 7 ilustra a evolução do desempenho em operações de ponto flutuante das GPUs comparada com CPUs nos últimos anos. A taxa de transferência de dados através do barramento da memória, ou largura de banda da memória, também teve um crescimento mais acelerado nas GPUs, conforme é apresentado na Figura 8.

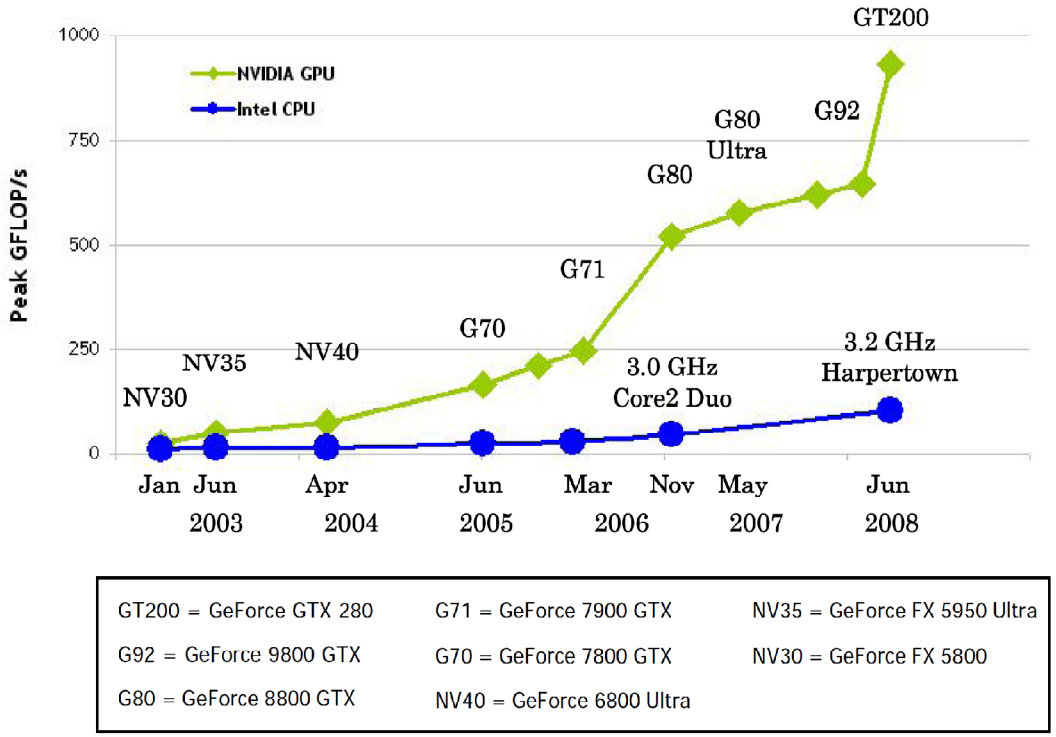

Figura 7 – Operações de Ponto Flutuante por Segundo (NVIDIA, 2009b).

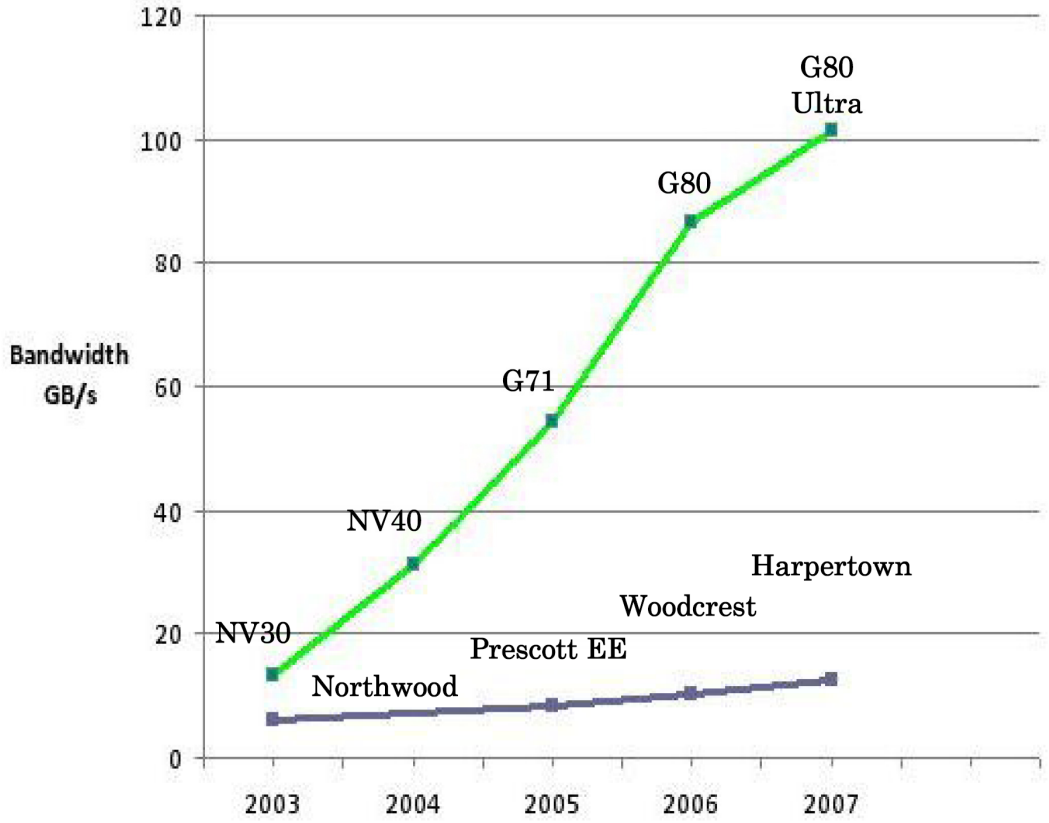

Figura 8 – Largura de Banda (NVIDIA, 2009b).

As GPUs, que eram originalmente projetadas para resolver o problema da renderização gráfica, seguiram um caminho de evolução que as tornaram especializadas para o processamento paralelo de dados. Uma maior quantidade de transistores é utilizada para processamento, reduzindo o espaço para controle de fluxo e memória *cache*. A CPU, por outro lado, apresenta um mecanismo de controle de fluxo mais sofisticado e maior área do chip dedicada a *cache*. Estas diferenças são ilustradas na Figura 9.

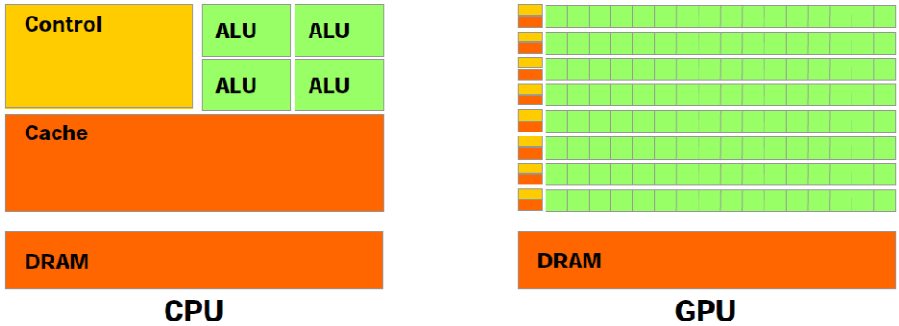

Figura 9 – A GPU dedica mais transistores para o processamento de dados (NVIDIA, 2009b).

Algumas das limitações do CUDA são: (i) impossibilidade de utilizar recursão e ponteiro para função; (ii) a latência de acesso à memória, ocasionada

pela falta de *cache*, pode ser minimizada pela divisão do problema de forma a aproveitar a capacidade de administrar um elevado número de *threads*. Enquanto um grupo de 32 *threads*, denominado *warp*, espera pelo dado a ser lido da memória, os recursos do hardware são usados para realizar operações em outros *warps*.

3.3. Elementos de Hardware da GPU

Os principais elementos do hardware da GPU são o Processador de Fluxo (SP), o Multiprocessador de Fluxo (SM) e os diferentes tipos de memórias disponíveis, os quais serão descritos em maiores detalhes nos próximos itens.

3.3.1. Processador e Multiprocessador de Fluxo

Um multiprocessador é constituído por oito processadores escalares, uma unidade de instrução, memória compartilhada interna ao chip e duas unidades especiais para execução de operações complexas, como seno, cosseno, exponencial e logaritmo.

Cada um dos oito processadores possui uma unidade de ponto flutuante de 32 bits, capaz de realizar operações de ponto flutuante e de inteiros, tais como multiplicação, adição, multiplicação-adição combinadas numa operação, subtração, comparação, etc. Geralmente uma placa de elevado nível possui vários multiprocessadores. Podemos citar como exemplo a Tesla C1060, a qual possui 30 multiprocessadores, totalizando 240 processadores.

O multiprocessador foi projetado para criar, administrar e executar *threads* concorrentes com um custo mínimo devido à *overhead* e sincronização. As barreiras de sincronização consomem uma única instrução. Uma instrução emitida pela unidade de instruções é executada por até 32 *threads*. Isso possibilita um paralelismo fino, sendo possível atribuir uma *thread* para cada elemento de dado, como por exemplo, atribuir uma *thread* para cada pixel de uma imagem (NVIDIA, 2009b).

3.3.2. Tipos de Memória da GPU

A memória de maior capacidade e também a de maior latência é a memória global, ou memória da placa. Esta memória aceita operações de leitura e escrita provenientes tanto da CPU, quanto da GPU, sendo a principal memória de troca de dados entre CPU e GPU. A latência de acesso a esta memória varia entre 400 e 600 ciclos de *clock*. Uma GPU Tesla C1060 possui 4 GB de memória RAM GDDR3.

A memória compartilhada é uma memória interna ao chip, o que a torna rápida, porém possui um tamanho bastante limitado de apenas 16 KB por multiprocessador. É útil para a troca de dados entre *threads* de um mesmo bloco. A latência pode ser de apenas um ou dois ciclos se forem empregadas técnicas adequadas de programação, evitando conflitos de bancos de memória.

A memória de textura é uma memória somente leitura para a GPU, devendo ser gravada pela CPU, que possui cerca de 6 a 8 KB de memória *cache* por multiprocessador. Ela é implementada no mesmo espaço físico da memória global e, portanto, em casos de falta de dados na memória *cache*, o tempo de latência é o mesmo da memória global. Possui ainda modos de endereçamento diferentes ou filtros para alguns formatos específicos de dados.

A memória de constantes é uma memória somente leitura com capacidade total de 64 KB e possui *cache*. A maneira mais rápida de acessar este tipo de memória ocorre quando todas as *threads* de um *half-warp* (grupo de 16 *threads*) precisam ler um mesmo elemento. Neste caso a latência será de um ou dois ciclos. Se for necessário fazer a leitura de múltiplos elementos, a latência será proporcionalmente maior, em relação ao número de elementos diferentes lidos pelas *threads* de um mesmo *half-warp* (NVIDIA, 2009b).

Uma GPU com capacidade de cálculo 1.3 possui 16384 registradores de 32 bits por multiprocessador. Considerando que neste tipo de GPU o número máximo permitido de *threads* ativas é de 1024 por multiprocessador, então teremos um mínimo de 16 registradores por *thread*. Os diferentes tipos de memória empregados numa GPU são apresentados na Figura 10.

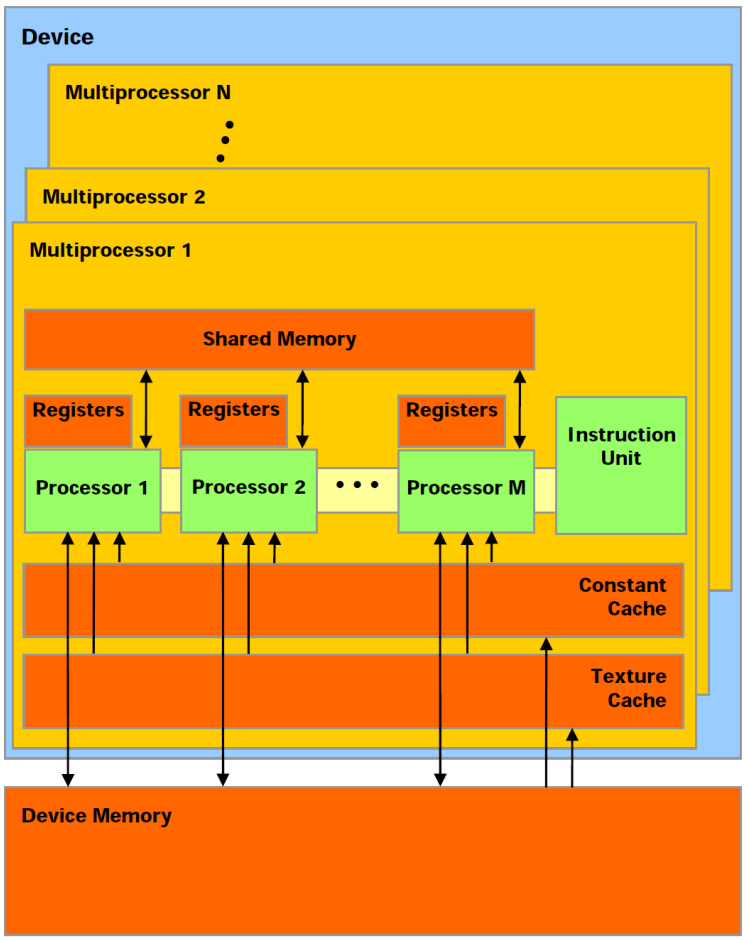

Figura 10 – Tipos de memória das GPUs da NVIDIA (NVIDIA, 2009b).

3.4. Elementos de Software da GPU

Um algoritmo empregando GPGPU consiste de um fluxo de controle rodando seqüencialmente numa CPU, operações de transferência de dados entre a memória da CPU e da GPU e chamadas de *kernel* (função escrita em CUDA para GPU) que executarão a tarefa usando o paralelismo de dados na GPU.

O primeiro passo do algoritmo de controle, rodando na CPU, é a chamada de funções para realizar alocações e transferências dos dados para a memória da GPU. As instruções CUDA são então enviadas para a GPU e executadas sobre os dados. Quando o *kernel* termina sua execução, é necessário realizar a transferência dos resultados para a memória da CPU, completando o processo ilustrado na Figura 11.

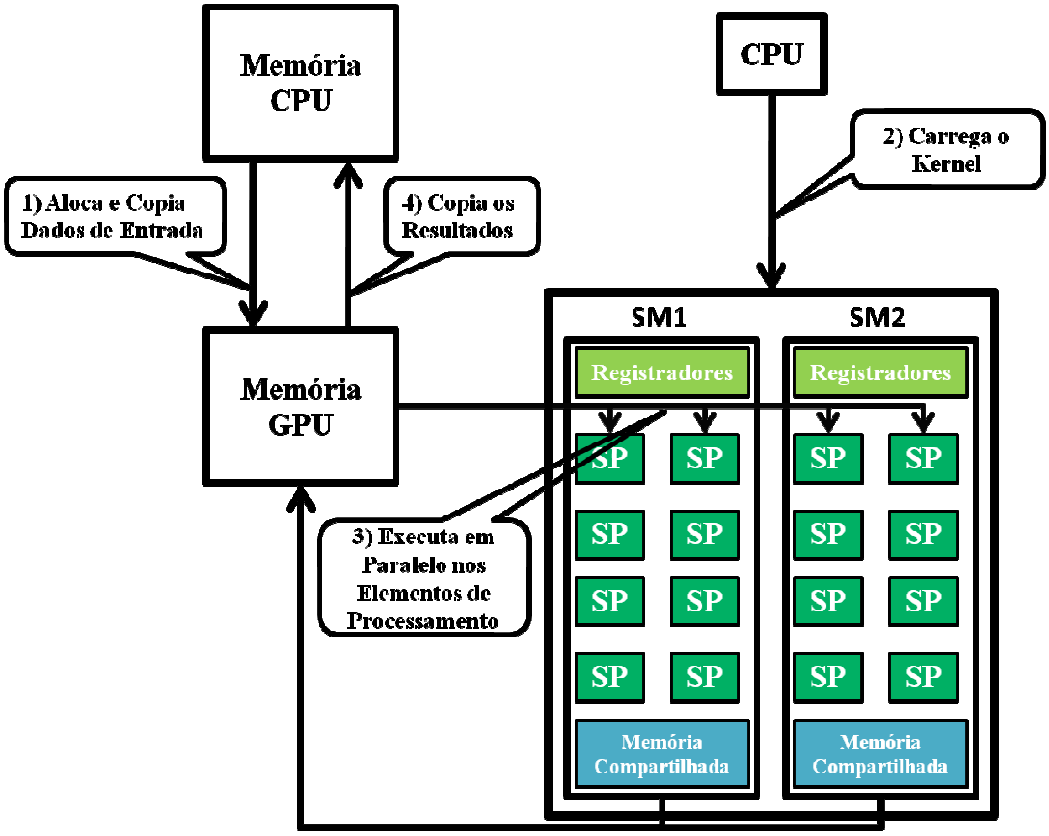

Figura 11 – Fluxo de processamento de um software empregando GPGPU.

A execução do *kernel* na GPU usando múltiplas *threads* é baseada numa hierarquia de três níveis. O *grid* é o nível mais alto e é constituído pelo conjunto de todas as *threads* executadas numa chamada de *kernel*. O *grid* é subdividido em blocos de *threads*, que por sua vez contém as *threads* no nível mais baixo.

3.4.1. Kernel

Uma função escrita para rodar na GPU é chamada de *kernel*. A diretiva de compilação *__global__* é indispensável na declaração de um *kernel* cuja chamada será realizada a partir do programa principal rodando na CPU. A diretiva *__device__* torna a função invisível para a CPU e somente poderá ser usada num *kernel* chamado a partir da GPU.

O programa principal faz a chamada de um *kernel* de maneira semelhante a qualquer função, exceto pela necessidade de incluir os parâmetros para configuração de paralelismo antes de passar as variáveis usando a notação $func\_kernel <<< a, b, Ns >>> (V1, V2 \ldots)$. O tamanho do *grid* em cada uma das três dimensões é especificado através de a, que é uma estrutura com os

campos $a.x$, $a.y$ e $a.z$. De forma semelhante, a configuração do tamanho dos blocos é realizada através de b. Ns é reservado para os casos em que se deseja especificar dinamicamente a quantidade de memória compartilhada a ser usada em cada bloco.

3.4.2. Grid

O nível mais alto da hierarquia de paralelismo de uma GPU é o *grid*, o qual é composto por todas as *threads* executando numa chamada de *kernel*. O *grid* é subdividido em blocos de *threads*, sendo limitado ao tamanho máximo de 65535 X 65535 X 1 blocos. O programa principal define explicitamente a quantidade de blocos e a sua distribuição no *grid* através da passagem de parâmetros ao *kernel*. A organização das *threads* em blocos e dos blocos no *grid* é ilustrada na Figura 12.

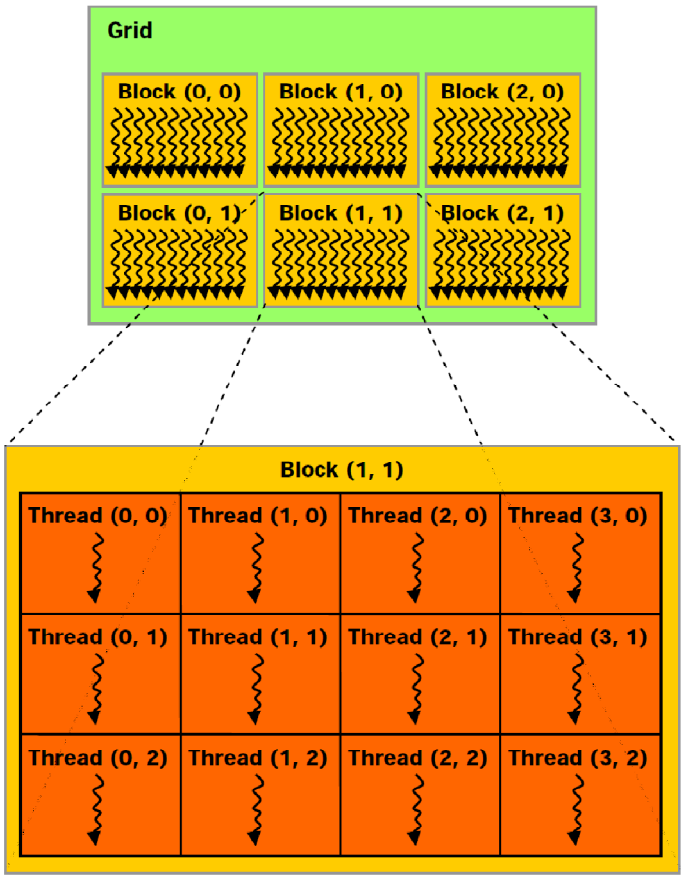

Figura 12 – *Grid* e Blocos de *Threads* (NVIDIA, 2009b).

3.4.3. Bloco de *Threads*

O segundo nível da hierarquia de *threads* é composto pela organização destas em blocos. Cada bloco é limitado a um total de 512 *threads* que podem ser distribuídas em três dimensões. De forma semelhante ao *grid*, o programa principal define dinamicamente a configuração dos blocos de *threads* através da passagem de parâmetros na chamada do *kernel*.

Os registradores de um multiprocessador são alocados para todas as *threads* de um bloco simultaneamente e somente serão liberados quando o bloco terminar de executar. Por isso é necessário um número relativamente grande de registradores por multiprocessador para que cada *thread* tenha um número razoável de registradores para sua execução. Um multiprocessador pode ativar e executar até oito blocos de *threads* simultaneamente, desde que o limite de 1024 *threads* ativas por multiprocessador não seja ultrapassado.

Quando as *threads* de um bloco são sincronizadas, o poder computacional do hardware da GPU pode ser empregado para executar operações em outro bloco independente. Assim, o custo de sincronização das *threads* dentro de um bloco é pequeno. A memória compartilhada pode ser acessada para leitura e escrita por todas as *threads* de um mesmo bloco, permitindo a colaboração e a troca de dados entre elas de forma rápida e eficiente. As *threads* de blocos diferentes só podem ser sincronizadas com segurança no término da execução do *kernel* e qualquer troca de dados só pode ser realizada através da memória global, a qual possui uma latência elevada.

3.4.4. Warp

A unidade de instruções do multiprocessador pode emitir uma instrução a cada dois ciclos. Os oito processadores que compõem o multiprocessador rodam no dobro da freqüência do resto da placa e podem iniciar uma instrução a cada ciclo. Assim, cada instrução emitida pela unidade de instruções pode ser executada por 32 *threads*, antes que a próxima instrução seja emitida, definindo o tamanho do *warp*. Para obter-se o máximo desempenho possível é importante

manter o número de *threads* de um bloco como sendo múltiplo do tamanho do *warp*, ou seja, múltiplo de 32.

O multiprocessador emprega um modelo de arquitetura chamado SIMT (*Single Instruction, Multiple Thread*). Esse modelo é muito semelhante ao modelo SIMD. Quando um bloco de *threads* é atribuído para ser executado num multiprocessador, o bloco é inicialmente dividido em *warps*. Todas as *threads* de um *warp* são inicializadas juntas, contudo elas são livres para divergir e executar independentemente. Assim, uma das principais diferenças do SIMT para o SIMD é que o controle da execução e dos desvios é especificado para cada *thread*. Contudo, para obter melhor desempenho é importante manter todas as *threads* de um mesmo *warp* executando a mesma instrução, evitando desvios de fluxo dentro do *warp.* Comandos condicionais que desviam o fluxo de todas as *threads* de um *warp* em relação a outro *warp* podem ser empregados sem redução de desempenho.

3.4.5. Escalabilidade

O modelo de escalabilidade empregado facilita a adaptação do código CUDA para a execução em GPUs diferentes. Quando o programa principal rodando na CPU chama um *kernel* que executará na GPU, os blocos de um *grid* são enumerados e distribuídos para os multiprocessadores com capacidade de execução disponíveis. Todas as *threads* de um mesmo bloco executam concorrentemente num único multiprocessador. Quando os blocos terminam, novos são atribuídos aos multiprocessadores que ficaram vazios. Isto permite que o mesmo código possa ser executado em duas GPUs, com números de multiprocessadores diferentes, sem ter que ser compilado duas vezes. O código automaticamente se adapta ao hardware disponível para extrair o máximo de desempenho, conforme o ilustrado na Figura 13.

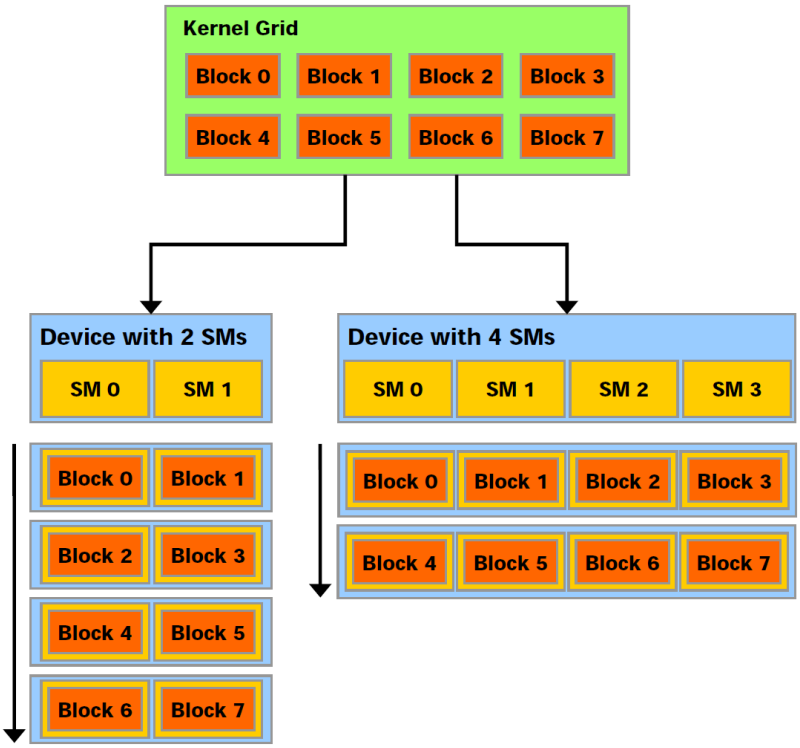

Figura 13 – Modelo de Escalabilidade da GPU (NVIDIA, 2009b).

3.5. Técnicas Otimizadas

Alguns cuidados devem ser levados em consideração ao realizarem-se leituras e escritas nos diferentes tipos de memória para que seja obtido o melhor desempenho possível com GPUs. As transferências de memória através do barramento PCI Express constituem um dos possíveis gargalos. A seguir serão descritos as formas mais eficientes de acesso às memórias global e compartilhada, bem como técnicas para redução do tempo total despendido com transferências de memória.

3.5.1. Acesso à Memória Global

A memória global de uma GPU pode ser vista em termos de segmentos alinhados de 16 e de 32 palavras de 32 bits. A Figura 14, adaptada de NVIDIA (2009a), ilustra a divisão da memória da GPU. Cada linha representa um segmento alinhado de 64 bytes, ou 16 números com precisão simples de ponto flutuante. Duas linhas da mesma cor representam um segmento alinhado de 128

bytes, ou 32 números com precisão simples de ponto flutuante. Na parte inferior são representados as 16 *threads* de um *half-warp*.

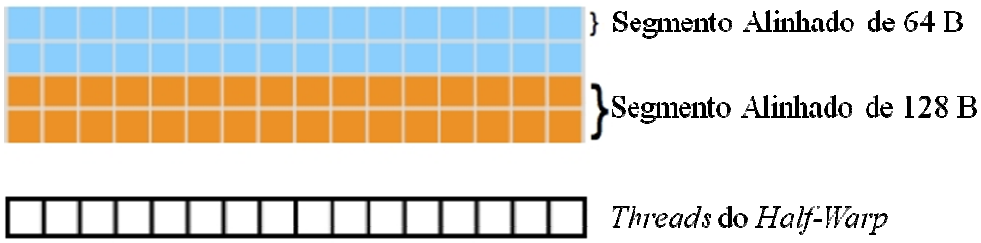

Figura 14 – Segmentos Alinhados de Memória e *threads* de um *half-warp*.

Uma GPU com capacidade de cálculo superior à 1.2, como a Tesla C1060, pode acessar a memória global realizando a leitura de blocos de dados de 8, 16 ou 32 palavras de 32 bits numa única operação. A operação de leitura pode atender todas as *threads* de um *half-warp* em paralelo. O tamanho do bloco de dados lido será o menor, dentro das opções 32, 64 ou 128 bytes, com capacidade para atender as *threads* de um *half-warp*. Assim, evita-se desperdício de largura de banda de memória.

O melhor desempenho é obtido quando as *threads* do *half-warp* acessam elementos de dados dentro de um mesmo segmento alinhado de 64 bytes. Nem todas as *threads* precisam participar na operação de leitura, mas é importante que nenhuma *thread* do *half-warp* acesse elementos fora do segmento alinhado de 16 palavras. A Figura 15 ilustra este padrão de acesso à memória global.

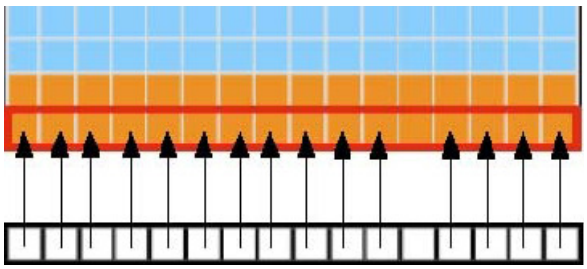

Figura 15 – As *threads* do *half-warp* acessam elementos da memória global dentro de um mesmo segmento alinhado de 16 palavras (NVIDIA, 2009a).

Para o padrão de acesso apresentado na Figura 15, será realizada uma única transferência de um bloco de dados de 64 bytes. Se houver permutação dos elementos acessados pelas *threads* do *half-warp*, dispositivos com capacidade de cálculo 1.1 terão que realizar 16 operações de transferência de dados. Porém, os

dispositivos com capacidade de cálculo 1.2 ou superior não terão seu desempenho afetado, continuando a realizar uma única operação de transferência de dados.

As *threads* do *half-warp* podem acessar 16 palavras de 32 bits que não estejam alinhadas com um bloco de 64 bytes da memória global, conforme o ilustrado na Figura 16. Isto gerará 16 transferências de dados para dispositivos com capacidade de cálculo 1.1. Para dispositivos com capacidade de cálculo 1.2 ou superior, poderá ser realizada uma única operação de transferência de dados de 128 bytes. Ou seja, 32 palavras são lidas da memória e somente metade delas será efetivamente usada. Contudo, o desempenho é melhor do que realizar 16 operações de leitura de blocos de 8 palavras, onde somente uma palavra por bloco será aproveitada.

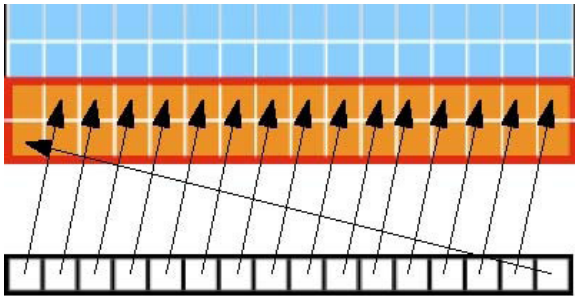

Figura 16 – Elementos não alinhados com o bloco de 16 palavras residem dentro do mesmo bloco de 128 bytes (NVIDIA, 2009a).

Os elementos não alinhados com o bloco de 16 palavras podem estar dentro de dois blocos diferentes de 128 bytes, conforme o ilustrado na Figura 17. Neste caso, os dispositivos com capacidade de cálculo 1.2 ou superior terão que realizar 2 operações de transferência de dados. Uma operação fará a leitura de 16 palavras das quais serão utilizadas 15 e a outra operação fará a leitura de 8 palavras, das quais serão utilizadas apenas uma.

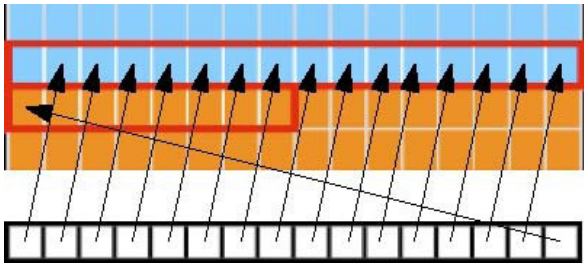

Figura 17 – Elementos não alinhados com o bloco de 16 palavras residem dentro de dois blocos diferentes de 128 bytes (NVIDIA, 2009a).

A Figura 18 apresenta a influencia das diversas formas de acesso à memória sobre o desempenho da largura de banda efetiva. São apresentados os casos para dois dispositivos, a NVIDIA GeForce GTX 280 com capacidade de cálculo 1.3 e

o NVIDIA Quadro® FX 5600 com capacidade de cálculo 1.0. Deslocamento unitário é o tipo de acesso ilustrado na Figura 16.

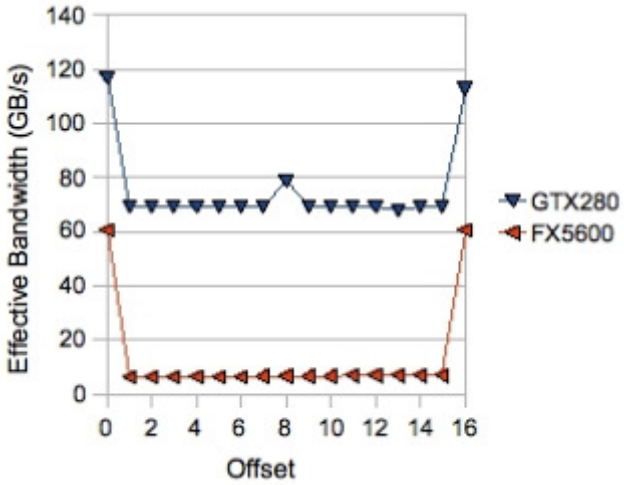

Figura 18 – Largura de banda efetiva para padrões de acesso à memória com deslocamento (NVIDIA, 2009a).

Pelo ilustrado na Figura 18, podemos verificar que, dispositivos com capacidade de cálculo 1.1 ou inferiores, tem seu desempenho de largura de banda drasticamente afetado por padrões de acesso com deslocamento diferente de zero ou 16. O desempenho da largura de banda para este caso ficou reduzido em cerca de oito vezes. Por outro lado, dispositivos com capacidade de cálculo 1.2 ou superior sofrem uma menor redução de desempenho. A redução na largura de banda para deslocamentos diferentes de zero ou 16 foi de apenas duas vezes.

As *threads* do *half-warp* podem acessar 16 elementos da memória deixando intervalos entre os elementos acessados conforme o ilustrado na Figura 19. Neste caso específico, todos os elementos lidos residem dentro do mesmo bloco de 128 bytes alinhados e um dispositivo com capacidade de cálculo 1.3 ou superior poderia realizar uma única operação de transferência de dados. Contudo, 32 palavras são lidas, sendo que somente 16 são efetivamente utilizadas.

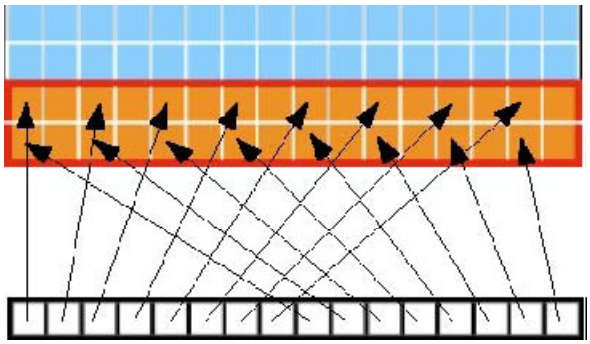

Figura 19 – Acesso a elementos da memória com intervalos entre os elementos acessados (NVIDIA, 2009a).

O padrão de acesso ilustrado na Figura 19 deve ser evitado. À medida que o número de intervalos entre os elementos lidos aumenta, a largura de banda efetivamente utilizada é drasticamente reduzida, conforme o ilustrado na Figura 20.

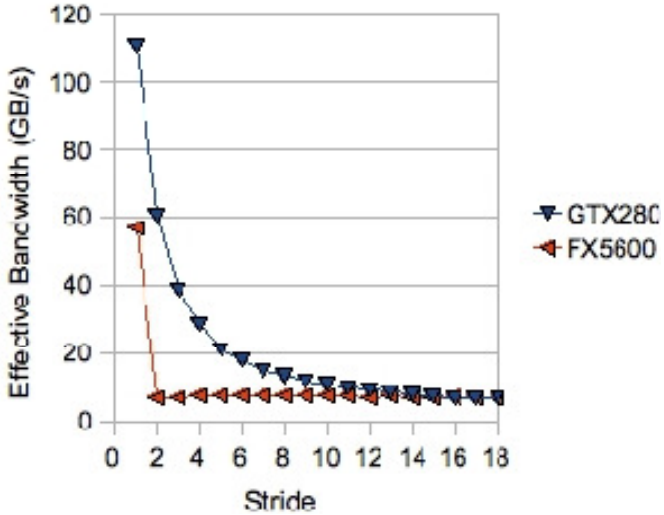

Figura 20 – Desempenho da largura de banda para padrões de acesso com intervalos entre os elementos lidos (NVIDIA, 2009a).

Um dos fatores responsáveis pela redução da largura de banda em acessos à memória com deslocamentos é o fato de que a memória global da GPU não possui *cache*. A memória de textura, apesar de ser somente leitura, possui *cache*. Assim, para o caso de variáveis que serão somente lidas, a memória de textura poderá ser uma alternativa para evitar a redução de desempenho em acessos com deslocamento diferente de zero ou 16. A Figura 21 compara o desempenho da largura de banda em leituras, com deslocamentos, para a memória global e para a memória de textura.

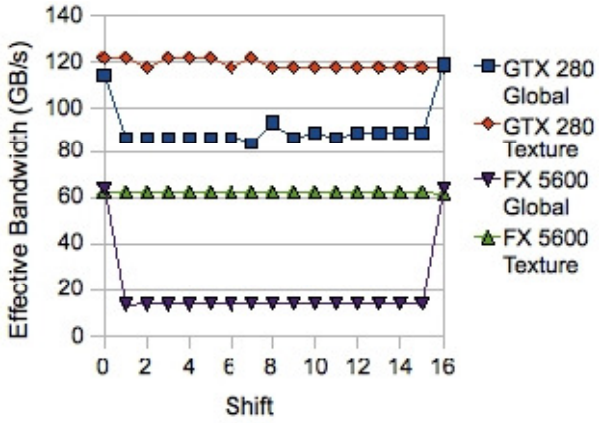

Figura 21 – Leitura, com deslocamentos, na memória global e na memória de textura (NVIDIA, 2009a).

3.5.2. Acesso à Memória Compartilhada

A memória compartilhada é interna ao chip e por isso é muito mais rápida do que a memória global. O acesso a memória compartilhada pode ser tão rápido quanto o acesso aos registradores, desde que não existam conflitos de banco entre as *threads* pertencentes a um *half-warp*.

A memória compartilhada é dividida em 16 módulos exatamente iguais, os quais são chamados de bancos. Uma operação de transferência de dados pode atender as 16 *threads* de um *half-warp* em paralelo. Para isso, é necessário que cada *thread* acesse um banco diferente, não importando à ordem dos acessos. Se duas *threads* acessarem um mesmo banco, então teremos um conflito de banco e serão necessárias duas operações de transferência para atender todas as *threads* de um *half-warp*. Assim, o tempo de acesso aumenta devido aos conflitos de banco.

Na Figura 22 são apresentados padrões de acesso à memória compartilhada sem conflitos de banco. No lado esquerdo temos um padrão de acesso linear e no lado direito temos um padrão de acesso com permutações randômicas entre as *threads* e os bancos. Em nenhum dos casos ocorre de duas *threads* usarem o mesmo banco e, portanto, não há conflitos de bancos, podendo todas as *threads* do *half-warp* serem servidas numa única operação de transferência.

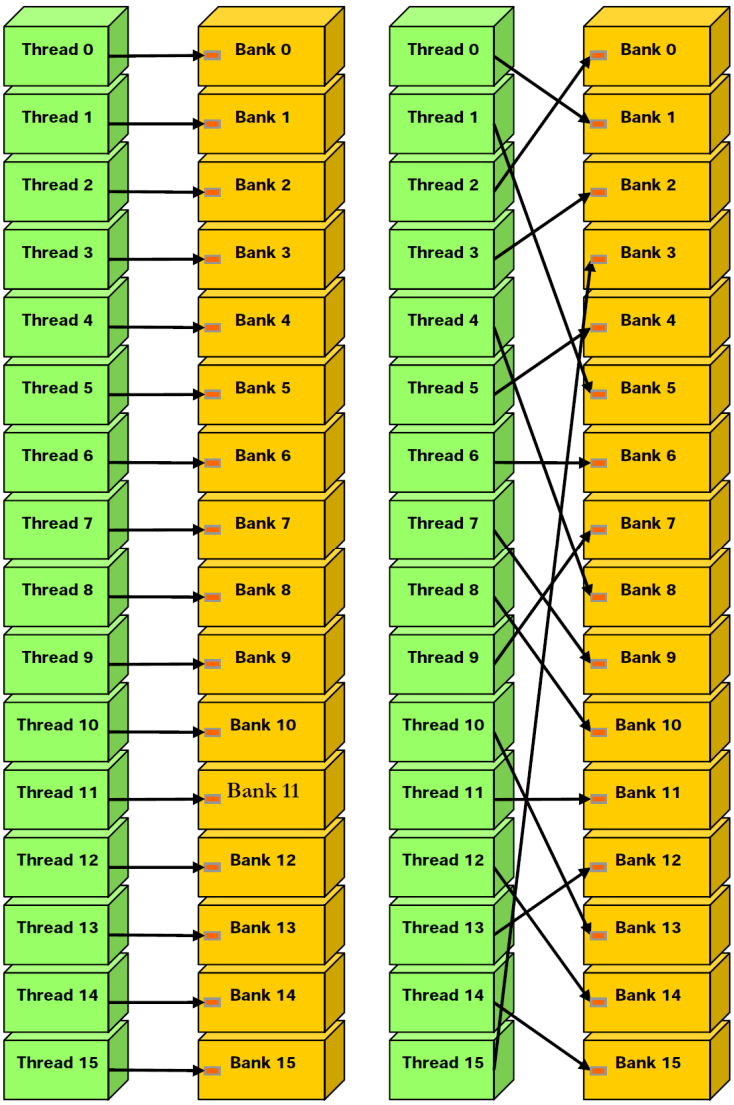

Figura 22 – Padrões de acesso à memória compartilhada sem conflitos de banco (NVIDIA, 2009b).

Padrões de acesso a memória compartilhada com conflitos de banco são ilustrados na Figura 23. No padrão da esquerda as 16 *threads* de um *half-warp* acessam 16 palavras de 32 bits com um intervalo entre cada elemento de dado. Dessa forma, todos os acessos recaem sobre somente oito bancos, gerando conflitos que necessitarão de duas operações de leitura. No padrão de acesso da direita, existe um intervalo de sete elementos vazios entre cada elemento acessado. Desta forma as leituras são concentradas em apenas dois bancos e serão necessárias oito operações de transferência de dados para atender todas as *threads*.

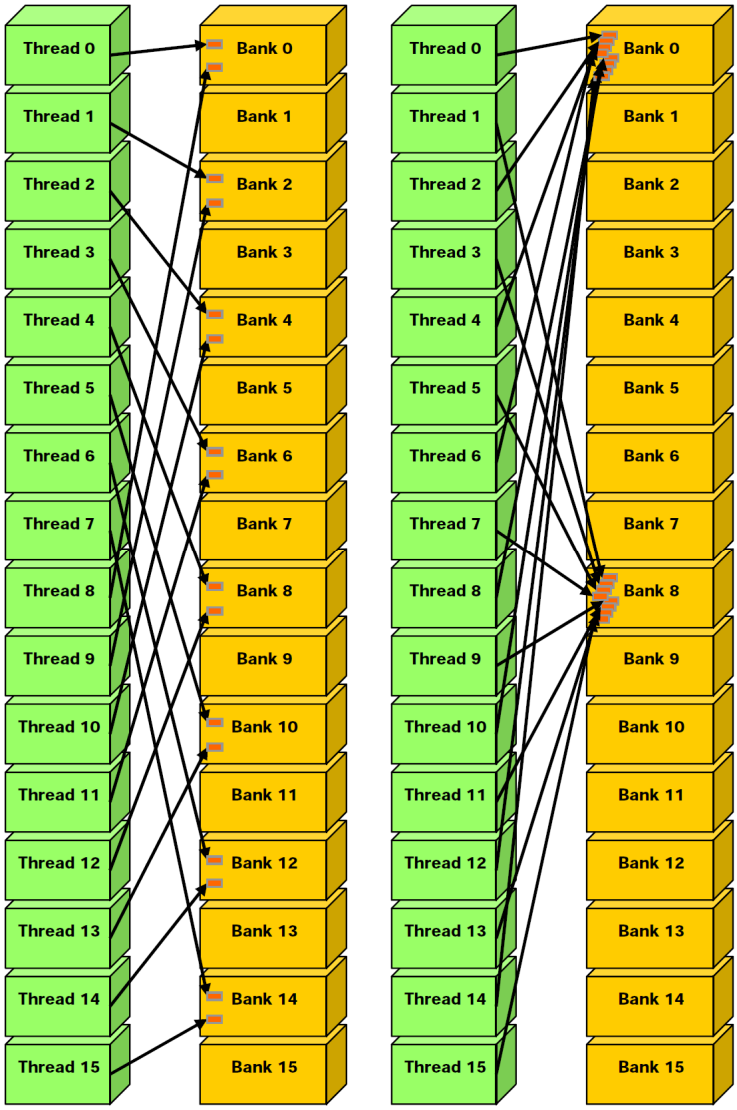

Figura 23 – Padrões de acesso à memória compartilhada com conflitos de banco (NVIDIA, 2009b).

3.5.3. Transferência de Dados Através do Barramento PCI Express

A maneira tradicional de realização de cálculos usando uma GPU consiste em transferir um vetor de dados da memória da CPU para a memória da GPU através do barramento PCI Express. Quando a transferência termina, um *kernel* é invocado para realizar operações sobre estes dados. Ao final da execução do *kernel*, os dados são novamente copiados da CPU para a GPU.

Contudo, em alguns casos, se as dependências entre os dados permitirem, poderá ser empregado um modelo onde os dados são copiados assincronamente. Assim, não é necessário esperar que todo o vetor de dados tenha sido transferido para a memória da GPU para que os cálculos comessem a ser realizados. Neste

caso são empregados múltiplos fluxos de processamento e à medida que os dados chegam à memória da GPU eles são processados e podem ser copiados de volta para CPU de forma assíncrona com a transferência de outras partes do vetor de dados (NVIDIA, 2009a). A Figura 24 ilustra os dois casos, mostrando a possibilidade de redução do tempo total pela sobreposição dos tempos de transferência e do tempo de execução.

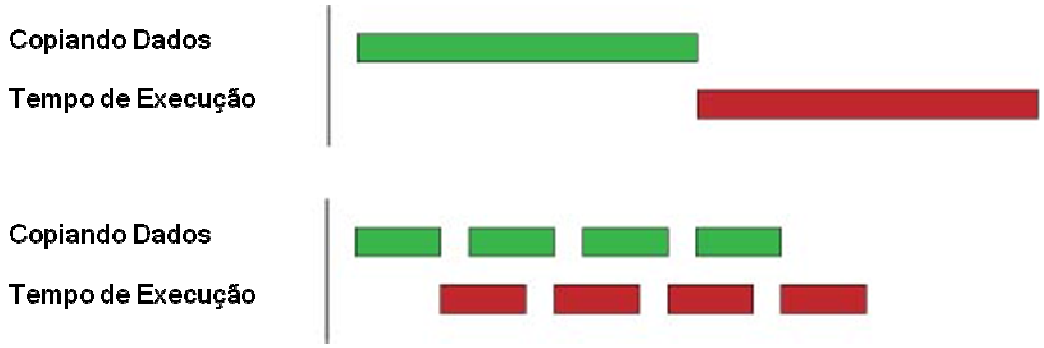

Figura 24 – Redução do tempo total através da sobreposição dos tempos de transferência e de execução (NVIDIA, 2009a).

3.5.4. Nível de Ocupação do Multiprocessador

Um multiprocessador de um dispositivo com capacidade de cálculo 1.2 ou superior pode manter até 32 *warps* ativos simultaneamente. Quando uma instrução executada por um Multiprocessador sobre um determinado *warp* apresenta um grande período de latência, os recursos de hardware podem ser aproveitados para executar outras instruções de outros *warps*. As instruções que possuem maior latência são os acessos de leitura e escrita na memória global.

O Nível de Ocupação do Multiprocessador é a razão entre o número de *warps* ativos e o número máximo possível de *warps* ativos. Um baixo Nível de Ocupação significa que o Multiprocessador não terá a possibilidade de executar instruções em outros *warps*, quando ocorrer uma instrução com grande latência. Desta forma, o desempenho é reduzido. Para dispositivos com capacidade de cálculo 1.2 ou superior recomenda-se manter um Nível mínimo de Ocupação de 18.75%. Contudo, melhores resultados podem ser obtidos com níveis mais elevados de Ocupação (NVIDIA, 2009a).

Por outro lado, um elevado Nível de Ocupação não necessariamente se traduz num elevado nível de desempenho. Por exemplo, aumentar o Nível de Ocupação de 66 % para 100 % não significa que o desempenho vá aumentar na

mesma proporção. Na prática, uma vez que o nível de 50 % tenha sido atingido, continuar aumentado o Nível de Ocupação não trará benefícios adicionais (NVIDIA, 2009a).