# 3
# Conceitos da Computação de Alto Desempenho

*A complexidade de custos de componentes mínimos tem aumentado a uma taxa de aproximadamente um fator de dois por ano...*

Gordon Moore

No capítulo anterior discutiu-se como é possível construir um software para um problema complexo em menos tempo e com maiores chances de sucesso. Neste capítulo, apresenta-se uma base teórica para se construir software priorizando seu desempenho e escalabilidade.

O termo Computação de Alto Desempenho (HPC – *High-performance Computing*) está, normalmente, associado à computação científica, pois essa área compreende cálculos maiores e mais complexos. Entretanto, devido ao tamanho e a complexidade cada vez maiores dos sistemas de informação atuais, as empresas começaram a investir em soluções com maior capacidade de memória e de processamento.

Com o início da substituição da válvula, largamente utilizada nas décadas de 1950 e 1960, pelo transistor, a capacidade de processamento começou a ser definida pela quantidade e pela velocidade desses novos componentes. Na década de 1970, a criação dos circuitos integrados e, subseqüentemente, dos multiprocessadores permitiu o aumento do número desses transistores juntamente com a redução dos custos de produção dos mesmos. Até então, não havia nenhuma previsão sobre o futuro do hardware e Gordon Moore, presidente da Intel na época, declarou que o número de transistores aumentaria a uma taxa de duas vezes por ano e que em 1975 o número desses componentes chegaria a 65.000 em um único chip. Essa afirmação se provou verdadeira até os últimos anos, mas tal taxa tem começado a diminuir nos últimos anos, devido aos custos, cada vez mais altos, para pesquisa de novos processadores (JOHNSON, 2009) e limitações físicas dos chips atuais (ADVE, ADVE, *et al.*, 2008).

Para se contornar o problema da dificuldade de aumento de desempenho dos computadores modernos, novos paradigmas têm sido estudados tanto em software

quanto em hardware. Por exemplo, a computação quântica (WIKIPEDIA, 2010), a computação paralela e computação reconfigurável (WIKIPEDIA, 2010).

Entre os novos paradigmas, os que apresentam os melhores resultados são aqueles que abordam o paralelismo no desenvolvimento tanto do software quanto do hardware. Entretanto, essa mudança na forma de se conceber um novo sistema, gera vários problemas. Por exemplo, ao se implementar um algoritmo usando uma solução paralela, é necessário saber quais partes desse algoritmo podem ser processadas ao mesmo tempo e quais partes não. Além disso, os desenvolvedores precisam identificar quais pontos precisam ser sincronizados para que não se gerem inconsistências. No caso da ANNCOM, apenas uma pequena parte do algoritmo de treinamento pode ser paralelizada, já que o modelo utilizado para treinar a rede neural é iterativo e os cálculos possuem dependência entre os passos.

Além das questões de software envolvidas na escolha por sistemas paralelos, existem várias outras implicações em hardware que precisam ser levadas em conta. Por exemplo, os processadores atuais possuem vários núcleos de processamento, mas com unidades de memória de acesso rápido (*Cache Memory*) independentes, o que, em termos de desempenho, em princípio, não seria a melhor solução. Entretanto, o sincronismo dessas unidades exige um hardware muito mais complexo e mais caro, por isso os projetistas optaram por unidades de memória separadas.

Por outro lado, os sistemas paralelos apresentam grandes vantagens, dentre as quais se destacam duas: a relação "operação por dólar" e a escalabilidade. A primeira, que contabiliza o custo de cada operação em um computador, vem se tornando cada vez maior, pois a indústria de hardware tem direcionado seus esforços para a computação paralela. A segunda representa a possibilidade de o paralelismo aumentar o poder de processamento de um sistema de acordo com a demanda, pois o software distribuirá suas tarefas pelas unidades de processamento independentemente do número de unidades.

## 3.1.
## Conceitos da Computação Paralela

Na seção anterior, foram discutidos quais fatores levaram os projetistas a optar pela utilização de sistemas paralelos e foram apresentadas algumas conse-

qüências dessa escolha. Nesta seção, se descreve como esses sistemas contornam o problema do limite de desempenho dos chips atuais explicando algumas abordagens apresentadas na literatura.

A Computação Paralela é uma forma de computação onde vários cálculos são realizados simultaneamente (ALMASI e GOTTLIEB, 1990), baseando-se no fato de que grandes problemas podem ser divididos em partes menores, e essas podem ser processadas simultaneamente. Nesse tipo de sistema, o paralelismo pode se dar em diversos níveis: de bit, de instrução, de dado ou de tarefa. Cada um desses modelos será discutido mais adiante.

A técnica de paralelismo já é empregada há vários anos, principalmente na computação de alto desempenho, mas recentemente, conforme mencionado anteriormente, o interesse no tema cresceu devido a várias limitações físicas que previnem o aumento de freqüência de processamento. Além disso, com o aumento da preocupação com o consumo de energia dos computadores, a computação paralela se tornou o paradigma dominante nas arquiteturas de computadores, sob forma de processadores com mais de um núcleo. O consumo de energia de um chip é dado pela Equação 1 (YOURDON, 1988).

$$P = VF \qquad \text{Equação 1}$$

Na Equação 1, $C$ é a capacitância, $V$ é a tensão e $F$ é a freqüência. Em 2004 a Intel cancelou seus modelos novos de processadores, *Tejas* e *Jayhawk*, por causa desse aumento de consumo em relação à freqüência de processamento. Esse episódio é normalmente citado como o fim da freqüência de processamento como paradigma predominante nas arquiteturas de computador (FLYNN, 2004).

### 3.1.1. Leis de *Amdahl* e *Gustafson*

Com a utilização de vários núcleos de processamento em um único chip, o aumento de velocidade com o paralelismo deveria ser idealmente linear, de forma que com $n$ núcleos, o tempo de execução se reduzisse $n$ vezes. Entretanto, muitos poucos algoritmos paralelos atingem essa situação, uma vez que a maioria deles fornece aumento de velocidade quase linear para poucos núcleos de processamento, tendendo a um valor constante para uma quantidade maior de elementos.

O aumento de velocidade potencial de um algoritmo em uma plataforma de computação paralela é dado pela Lei de Amdahl (AMDAHL, 1967), formulada por Gene Amdahl na década de 1960. Essa lei afirma que uma pequena parte do programa, que não pode ser paralelizada, limitará o aumento de velocidade geral que o paralelismo poderia oferecer. Os algoritmos consistem, normalmente, de partes paralelizáveis e partes seqüenciais. A relação entre os dois é dada pela Equação 2.

$$S = \frac{1}{(1-P)} \qquad \text{Equação 2}$$

Na Equação 2, *S* é o aumento de velocidade do programa e *P* é a fração paralelizável. Se a parte seqüencial de um programa representar 10% do tempo de execução, não se poderá obter mais do que dez vezes de aumento de velocidade, não importando o número de processadores, o que tornará inútil a adição de mais unidades de execução. Na Figura 7, a curva azul ilustra o aumento linear que um programa teria no caso ideal, enquanto a curva rosa indica o aumento real. Da mesma forma, a curva amarela indica o tempo de execução no caso ideal, enquanto a curva vermelha indica o tempo de execução real.

Uma representação gráfica é apresentada na Figura 8, assumindo que uma tarefa possui duas partes independentes: *A*, que ocupa cerca de 70% de todo tempo de execução, e *B*, que ocupa o restante. Nesse exemplo, se um programador tornar a parte *B* cinco vezes mais rápida, o tempo total de execução será reduzido em mais de 20%. Por outro lado, com menos esforço pode-se tornar a parte *A* duas vezes mais rápida, o que reduzirá o tempo total de execução em mais de 30%.

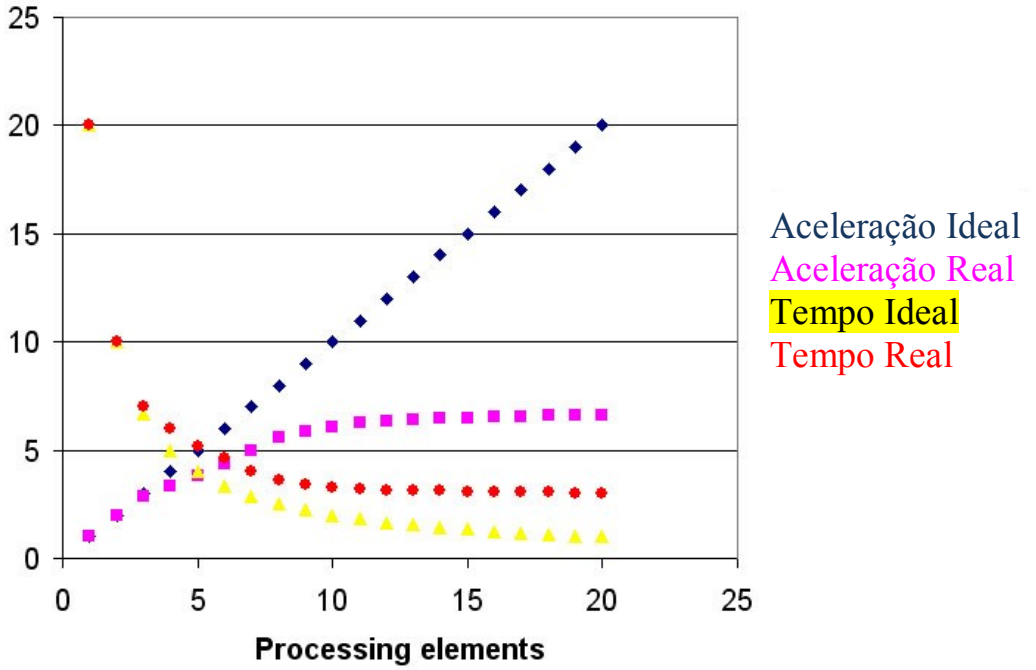



Figura 7 – O tempo de execução e o aumento de velocidade de um programa com paralelismo.

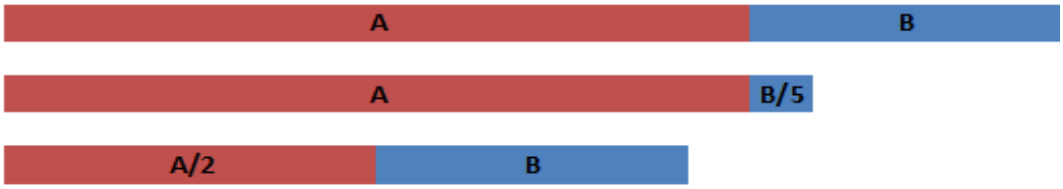



Figura 8 – Representação gráfica da lei de Amdahl.

A Lei de Gustafson (GUSTAFSON, 1988) também relaciona aumento de velocidade com fração paralelizável, mas não assume um tamanho de problema fixo e nem que o tamanho da parte seqüencial é independente do número de processadores. A Equação 3 formula a Lei de Gustafson, onde $P$ é o número de processadores, $S$ é o aumento de velocidade e $\alpha$ é a parte não paralelizável do processo.

$$S(P) = P - \alpha\ (P - 1) \qquad \text{Equação 3}$$

### 3.1.2. Tipos de Paralelismo

Uma das primeiras metodologias de classificação para computadores e programas paralelos e seqüenciais foi criada por Michael J. Flynn (FLYNN, 2004).

Atualmente conhecida como taxonomia de Flynn, ela classifica os programas e computadores por quantidade de fluxos de instruções e por quantidade de dados usada por tais instruções. A Tabela 2 apresenta as quatro classes propostas por Flynn, onde a classe SISD representa um sistema completamente seqüencial e o MIMD representa um dos sistemas paralelos mais usados atualmente. Um exemplo de um sistema MIMD é a computação distribuída, que consiste de várias máquinas ligadas em rede e cujo objetivo é realizar uma tarefa em comum.

| | Única Instrução (*Single Instruction*) | Múltiplas Instruções (*Multiple Instruction*) |
|---|---|---|
| Único Dado (*Single Data*) | SISD | MISD |
| Múltiplos Dados (*Multiple Data*) | SIMD | MIMD |

Tabela 2 – Taxonomia Flynn.



Além da taxonomia Flynn, pode-se classificar os sistemas paralelos segundo o nível em que se dá o paralelismo: (a) no bit; (b) na instrução; (c) no dado; (d) na tarefa.

A partir do advento da tecnologia de fabricação de chip VLSI na década de 1970 até cerca de 1986, o aumento da velocidade dos computadores era obtido dobrando-se o tamanho da palavra, que determina a quantidade de informação que pode ser processada por ciclo (CULLER, SINGH e GUPTA, 1998). Aumentando o tamanho da palavra, se reduz a quantidade de instruções que um processador deve executar para realizar uma operação em variáveis cujo tamanho é maior do que o da palavra. Historicamente, microprocessadores de quatro bits foram substituídos pelos de oito, depois pelos de dezesseis e mais tarde pelos de trinta e dois bits. A partir de então, o padrão 32-bit se manteve na computação de uso geral por duas décadas. Em 2003, a arquitetura 64-bit começou a ganhar mais espaço.

Considerando que, em sua essência, um programa de computador é um fluxo de instruções executadas por um processador, e que a execução de cada instrução exige diversas sub-tarefas, tais como busca, decodificação, cálculo e armazenamento de resultados, pode-se promover o paralelismo de tais sub-tarefas. Esse método é conhecido por paralelismo em instrução ou técnica de *pipeline*. Avanços nessa técnica dominaram as arquiteturas de computadores de meados da década de 1980 até meados da década de 1990 (CULLER, SINGH e GUPTA, 1998). Pro-

cessadores modernos possuem *pipelines* com 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 em dado é inerente a laços de repetição, focando em distribuir as iterações, ou o processamento de diversos dados, por diferentes nós computacionais para serem executadas em paralelo. Diversas aplicações científicas e de engenharia apresentam esse tipo de paralelismo. Uma dependência por laço é a dependência de uma iteração do laço com a saída de uma ou mais iterações anteriores, uma situação que impossibilita a paralelização de laços. Por exemplo, na ANNCOM, o treinamento é iterativo e cada época depende do resultado da anterior. Com o aumento do tamanho de um problema, geralmente aumenta-se a quantidade de paralelismo em dado disponível (CULLER, SINGH e GUPTA, 1998).

O paralelismo em tarefa é a característica de um programa paralelo em que diferentes cálculos são realizados no mesmo ou em diferentes conjuntos de dados (CULLER, SINGH e GUPTA, 1998). Isso contrasta com o paralelismo em dado, em que o mesmo cálculo é feito em diferentes conjuntos de dados. Paralelismo em tarefa geralmente não é escalável com o tamanho do problema (CULLER, SINGH e GUPTA, 1998). As placas gráficas atuais da NVIDIA adotaram uma abordagem conhecida como SIMT (*Single Instruction Multiple Thread*), e que está entre o paralelismo de tarefas e o paralelismo de dado, uma vez que a GPU executa várias *threads* ao mesmo tempo ao invés de uma *thread* com uma palavra grande. A tecnologia de placas gráficas será explicada na próxima seção.

## 3.2. Os Processadores Gráficos e a GPGPU

Até agora foram apresentadas algumas arquiteturas evolvendo vários processadores ou vários computadores. Entretanto, com a necessidade de cálculos muito complexos e soluções em tempo real, surge uma demanda por sistemas com tamanho grau de paralelismo que restringe a utilização das abordagens anteriores. O principal motivo disso é o custo financeiro e ambiental de se construir e se manter um espaço com vários computadores, os quais precisam ser mantidos em am-

bientes controlados e gerenciados por grandes sistemas e mão de obra especializada.

Ao passo que essa necessidade de poder computacional crescia, as placas gráficas estavam se tornando poderosos computadores, extremamente paralelizados, por causa da indústria do entretenimento digital e sua demanda por gráficos em alta definição. As Figura 9 e Figura 10 mostram um comparativo entre alguns processadores e algumas placas de vídeo, com relação ao desempenho e o acesso à memória.

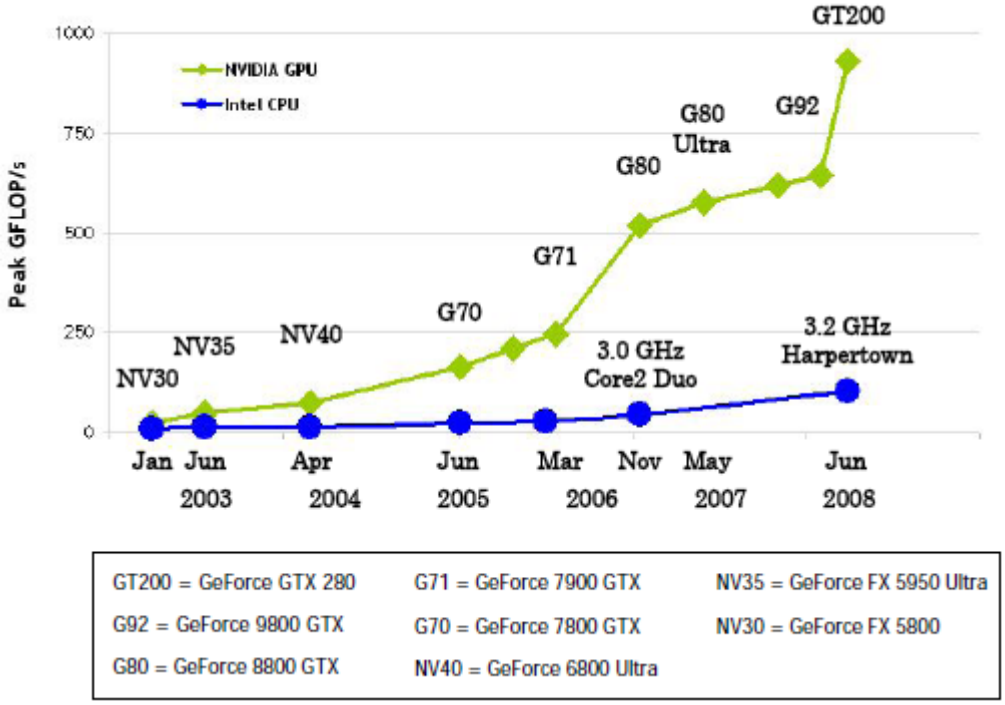



Figura 9 – Comparativo entre CPU e GPU com relação às operações de ponto flutuante por segundo (NVIDIA, 2009).

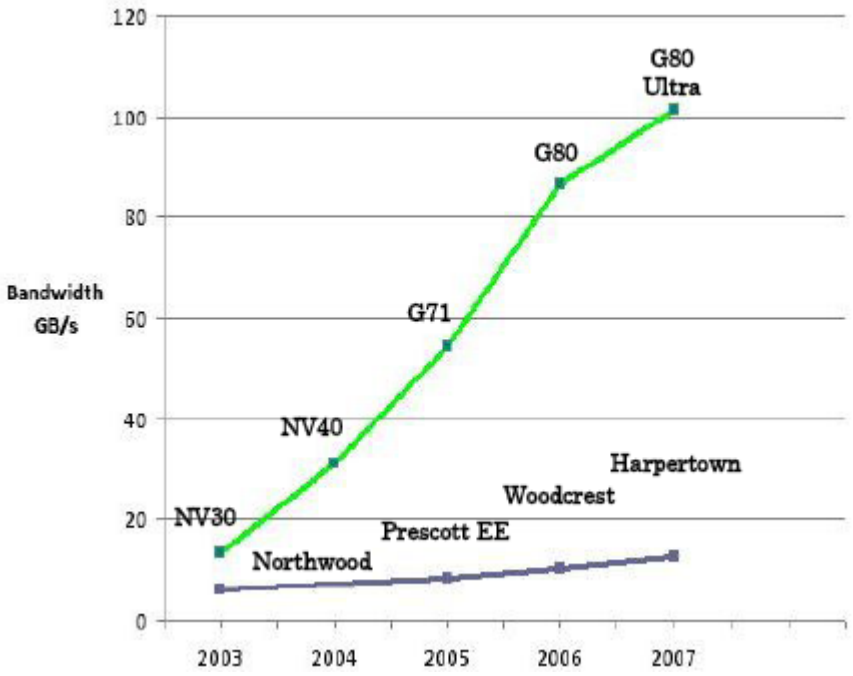



Figura 10 – Comparativo entre CPU e GPU com relação à largura de banda de memória memória (NVIDIA, 2009).

A razão da enorme discrepância entre os processadores convencionais e os processadores gráficos é que uma GPU (*Graphics Processing Unit*), ao contrário da CPU, utiliza muito mais transistores para processamento do que para controle de fluxo ou memória de acesso rápido (NVIDIA, 2009), conforme é ilustrado na Figura 11. Percebe-se que essa arquitetura se adapta melhor a problemas cujos dados possam ser quebrados em partes menores e processados de forma paralela, mas simples, como cálculos aritméticos, para que o mesmo programa seja executado nas pequenas partes ao mesmo tempo, dispensando, dessa forma, um controle de fluxo sofisticado. Além disso, como existem muitos cálculos aritméticos a serem executados, a latência de acesso à memória fica encoberta pelo tempo dos cálculos, dispensando o uso de grandes memórias de acesso rápido (NVIDIA, 2009).

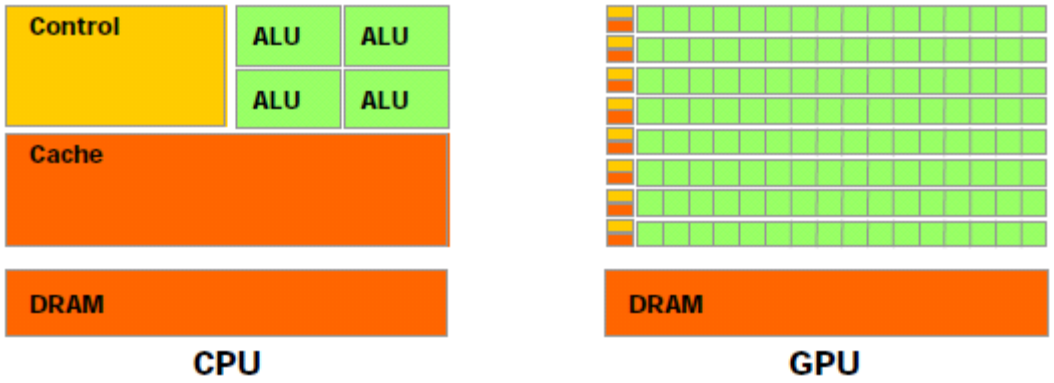



Figura 11 – Diferença entre GPU e CPU com relação à utilização de transistores (NVIDIA, 2009).

A tecnologia de chips gráficos se mostrou bem interessante para resolver alguns dos problemas antes aceitos como insolúveis considerando a tecnologia atual, além da possibilidade de acelerar outros. A utilização de um ou mais processadores gráficos para executar tarefas tradicionalmente feitas pelos processadores comuns ficou conhecida como Computação de Propósito Geral em Processadores Gráficos (*General-Purpose Computing on Graphics Processing Units* – GPGPU).

## 3.3. CUDA – NVIDIA

Em novembro de 2006, a NVIDIA lançou o CUDA, uma arquitetura para computação paralela de propósito geral que utiliza as placas gráficas da NVIDIA. A tecnologia CUDA oferece um ambiente que permite o desenvolvimento em linguagem C, C++, FORTRAN, OpenCL e, futuramente, DirectX Compute, conforme mostra a Figura 12 (NVIDIA, 2010).

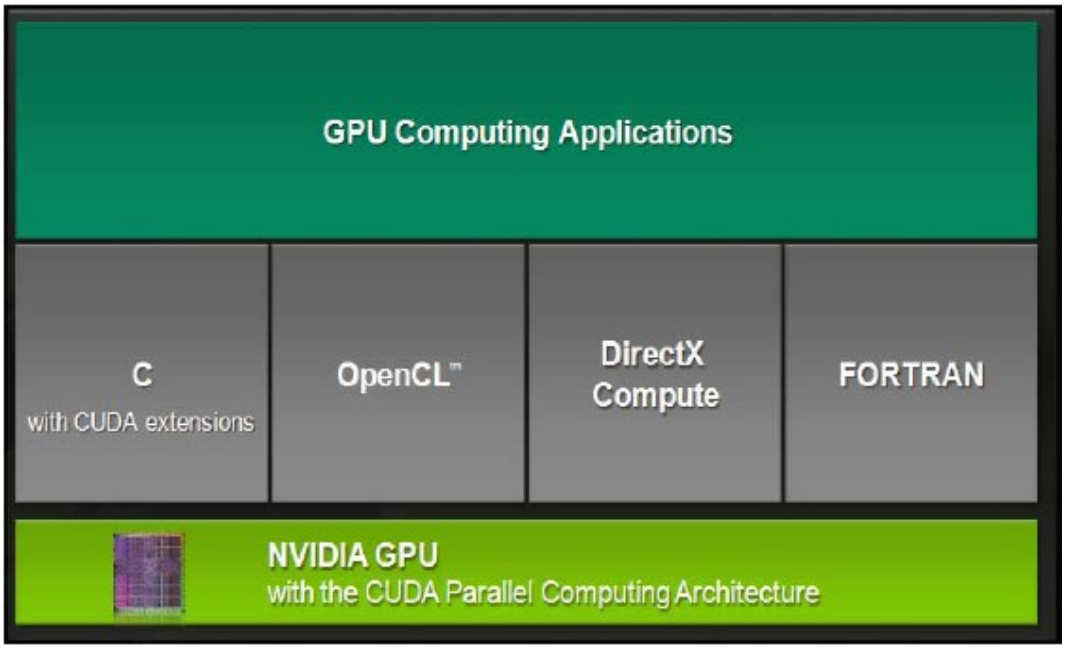



Figura 12 – Suporte de CUDA para várias linguagens (NVIDIA, 2009).

A arquitetura da NVIDIA possui três níveis de abstração: a hierarquia de grupos de *threads*, as memórias compartilhadas e as barreiras de sincronização. Isso torna a curva de aprendizado da linguagem relativamente pequena e requer poucas extensões para as linguagens de programação. Além disso, as abstrações provêem paralelismo tanto no dado quanto nas *threads*, necessitando, por parte do programador, apenas de uma simples divisão de dados e tarefas (NVIDIA, 2009).

Outra vantagem das abstrações é que elas ajudam o desenvolvedor a dividir o problema em tarefas menores, que podem ser resolvidas de forma independente e, conseqüentemente, paralela. Tal decomposição é feita de modo que as *threads* possam cooperar entre si ao resolver as sub-tarefas e, ao mesmo tempo, possibilitar a escalabilidade, uma vez que as *threads* podem ser agendadas para serem resolvidas em qualquer núcleo disponível. Um programa compilado em CUDA pode, nesse caso, ser executado em uma máquina independentemente do número de processadores, o qual será verificado em tempo de execução (NVIDIA, 2009). Dessa forma, o modelo de programação permite à arquitetura abranger vários tipos de máquinas, com placas variando de algumas dezenas até milhares de núcleos.

### 3.3.1.
### O Modelo de Programação

Nesta seção, apresentar-se-á o modelo de programação da arquitetura CUDA. Conforme mencionado na seção 3.3, a arquitetura CUDA fornece algumas extensões para a linguagem C. Elas permitem que o programador invoque funções, chamadas *kernels*, que serão executadas *N* vezes em *N threads* diferentes. Cada *thread* possui um identificador único permitindo que o desenvolvedor acesse cada uma delas.

As *threads* são hierarquizadas em blocos tridimensionais (a Figura 13 faz uma representação bidimensional para simplificar a visualização) e esses são organizados em grades de duas dimensões – ver Figura 13.

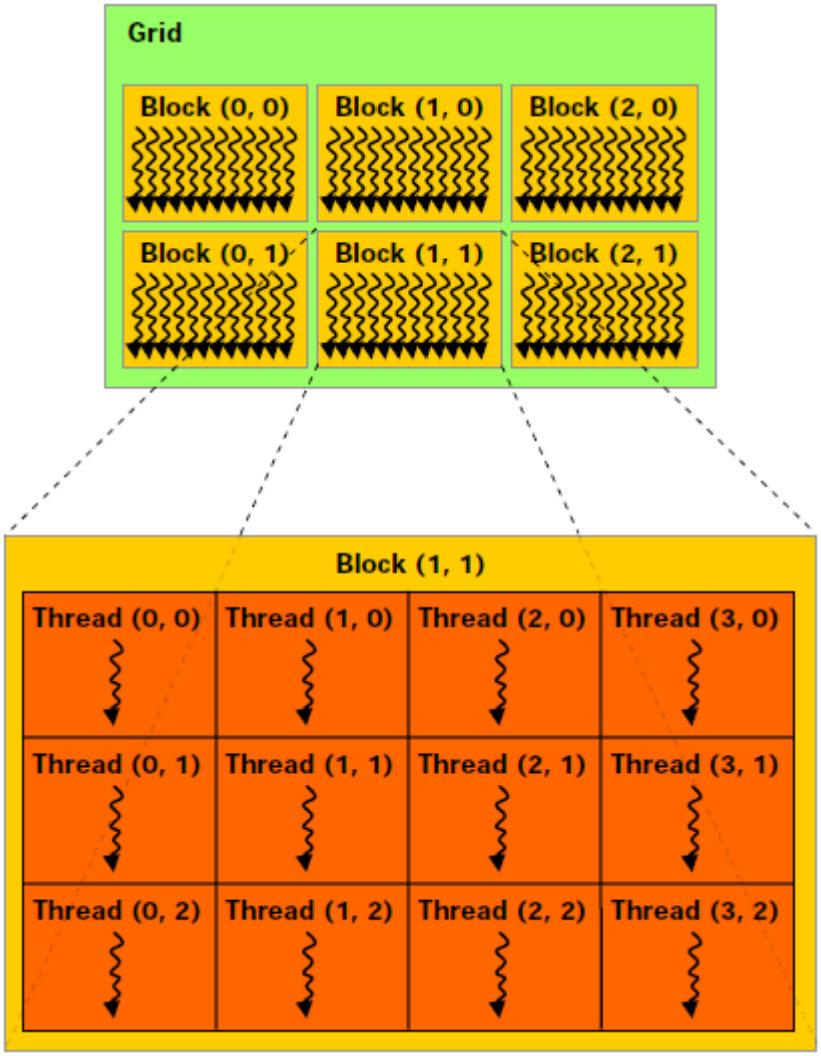



Figura 13 – Grade de blocos de *threads* (NVIDIA, 2009).

Isso torna mais fácil o entendimento ao se computar elementos vetoriais e matriciais, uma vez que os índices das *threads* são seqüenciais dentro dos blocos. Por exemplo, num espaço de duas dimensões de tamanho $(D_x, D_y)$, a posição $(x, y)$ terá o índice $(x + yD_x)$, e se o espaço for de três dimensões com tamanho $(D_x, D_y, D_z)$, a posição $(x, y, z)$ terá o índice $(x + yD_x + zD_xD_y)$.

As *threads* de um mesmo bloco podem cooperar entre si através do sincronismo de suas execuções e de uma memória compartilhada (*shared memory*). Para funcionar de forma eficiente, a memória compartilhada possui baixa latência e fica próxima dos núcleos, podendo ser comparada com a memória L1 de uma CPU. A única restrição ao uso do bloco de *threads* e sua memória compartilhada é seu tamanho limitado. Entretanto, é possível dividir todo o *kernel* em blocos iguais e organizá-los numa grade para que, o máximo possível deles, seja executado ao mesmo tempo pela placa. Nesse caso, o número de *threads* de uma função a ser executada será o número de *threads* por bloco vezes o número de blocos.

Além da memória compartilhada, a placa de vídeo possui a memória global, que pode ser acessada pelas *threads*. Essa, entretanto, é compartilhada com todo o programa, ao invés de apenas dentro dos blocos, e possui duas áreas somente para leitura, destinadas a constantes e a texturas. A última pode ser mapeada através de mapas de textura. A Figura 14 apresenta os diferentes níveis de memória existentes na placas gráficas da NVIDIA. Tal hierarquia de memória resulta do fato do modelo de programação CUDA assumir que as *threads* são executadas em um dispositivo fisicamente separado do computador, como um co-processador. Por exemplo, se um programa possuir partes em *kernel* e partes normais, a primeira será executada na GPU e a CPU executará a segunda.

O modelo de programação CUDA assume que tanto o computador quanto a placa gráfica possuem suas próprias memórias chamadas, respectivamente, de *host memory* e *device memory*. Com isso, o programa precisa gerenciar a memória global, a de constantes e a de texturas, além da memória do computador e as transferências entre computador e placa de vídeo.

As características da arquitetura permitem que sejam desenvolvidos programas chamados heterogêneos – seqüenciais e paralelos – que são executados ao mesmo tempo pela GPU e pela CPU. A Figura 15 ilustra esse tipo de programação.

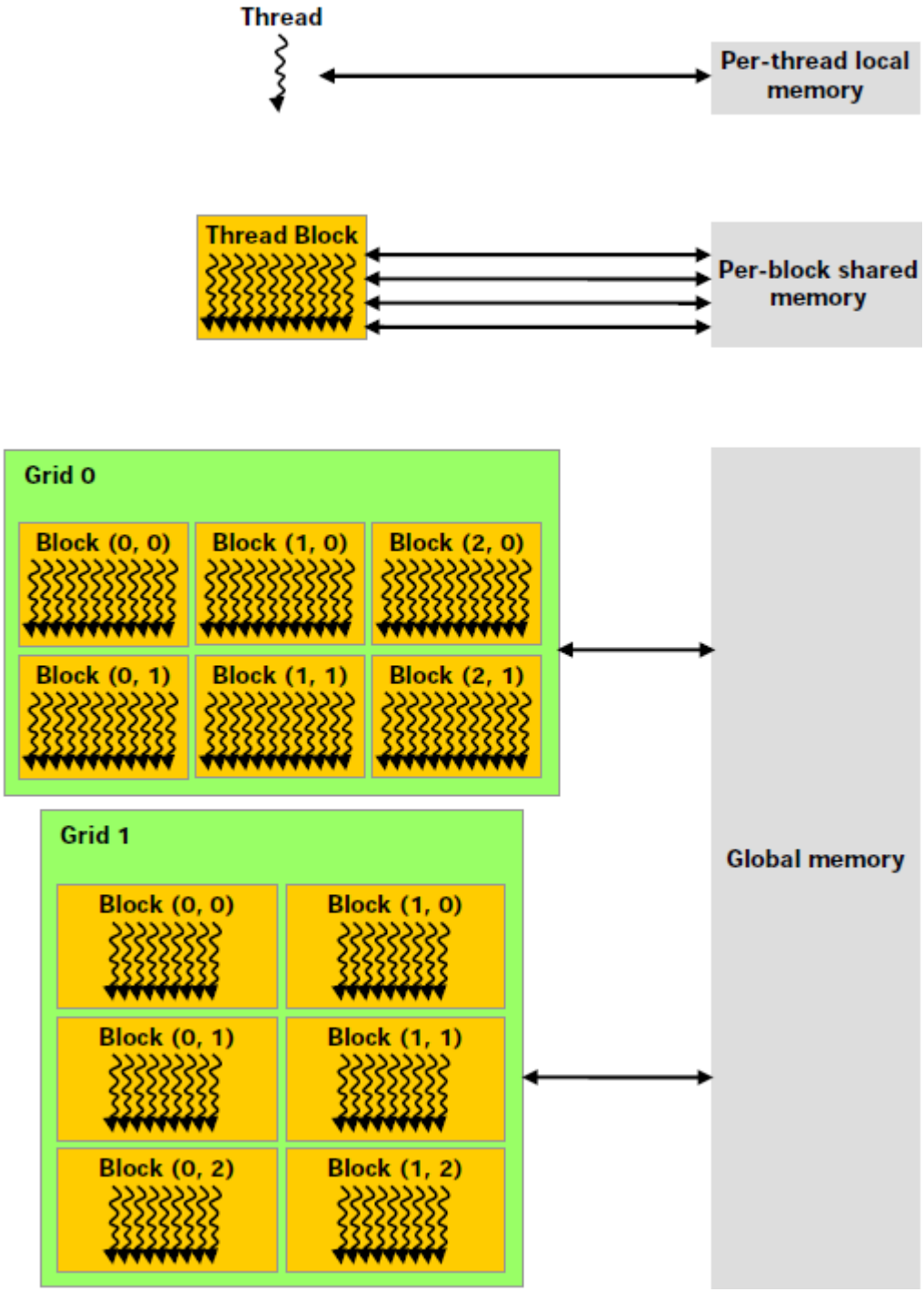



Figura 14 – Os diferentes níveis de memória nas placas de vídeo da NVIDIA (NVIDIA, 2009).

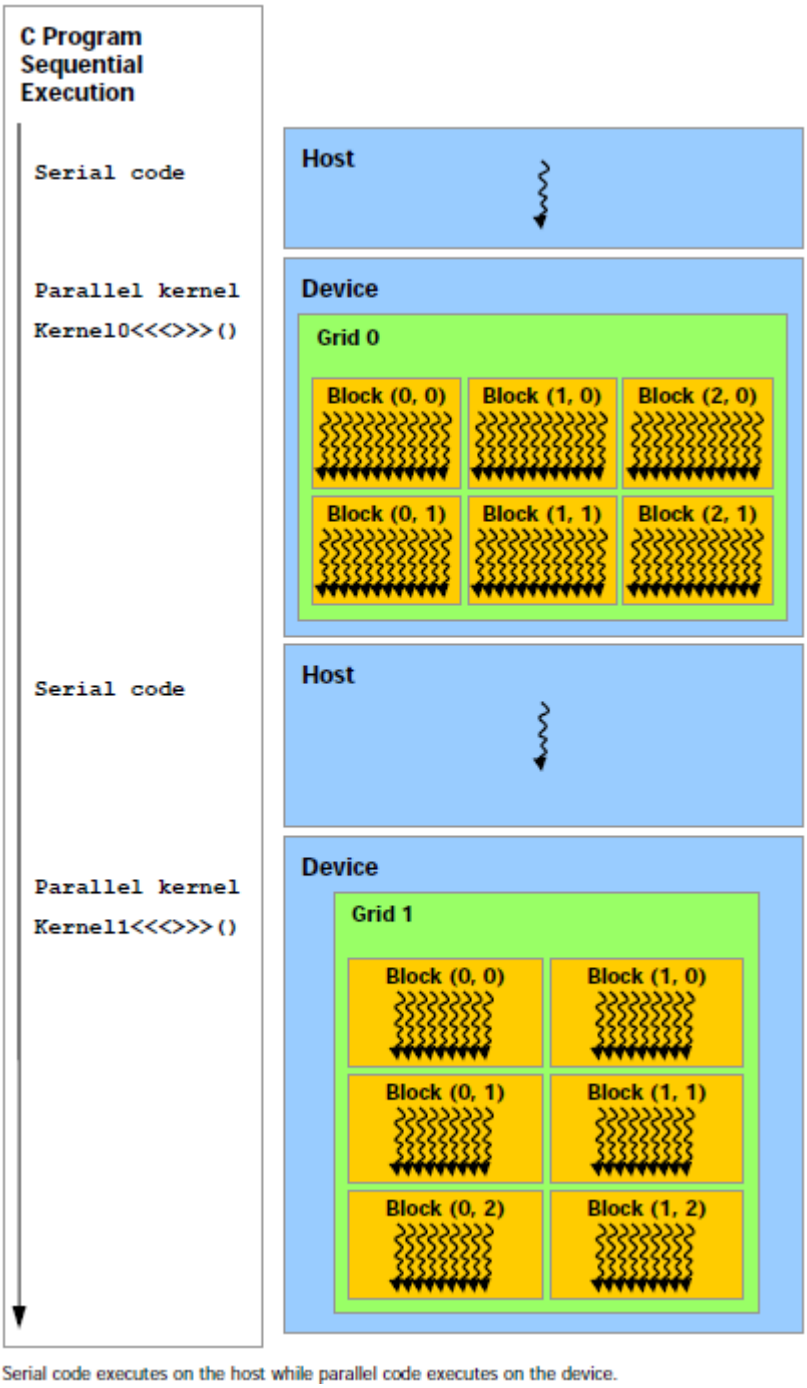



Figura 15 – Modelo de programação heterogêneo (NVIDIA, 2009).

A arquitetura CUDA suporta, atualmente, duas interfaces para se codificar programas: "*C for CUDA*" e "*CUDA Driver API*". Elas são exclusivas e não se pode misturá-las em um mesmo programa. A primeira interface vem com uma API (*Application Programming Interface*) usada em tempo de execução e adiciona apenas algumas instruções à linguagem C. Já a segunda é uma API em C, que fornece funções para carregar, como módulos, os binários CUDA, verificar seus parâmetros e executá-los. Vale salientar que o código escrito em *C for CUDA* é

mais conciso, mais simples e suporta depuração, uma vez que é montado diretamente sobre a outra interface. Entretanto, códigos usando *CUDA Driver API* oferecem maior nível de controle e são independentes de linguagem, uma vez que podem trabalhar diretamente com binários ou código *assembly* (NVIDIA, 2009).

### 3.3.2. A Implementação do Hardware

Na seção anterior discutiu-se o modelo de programação e as interfaces da arquitetura CUDA. Para concluir a apresentação da tecnologia de computação de propósito geral da NVIDIA, descreve-se o hardware das placas de vídeo que suportam a tecnologia.

A arquitetura CUDA foi construída sobre uma matriz de multiprocessadores (*Streaming Multiprocessors – SM*) (NVIDIA, 2009). Ao se invocar um *kernel* (dividido em uma grade) em CUDA numa CPU, os blocos de *threads* são enumerados e distribuídos para os multiprocessadores disponíveis conforme mostra a Figura 16. As *threads* de um bloco são executadas, de forma concorrente, em um multiprocessador, e quando esse bloco é finalizado, outro bloco é inicializado até que toda a grade seja finalizada. Mais do que isso, se uma placa gráfica possuir *N* multiprocessadores, ela poderá executar até *N* blocos ao mesmo tempo.

Os multiprocessadores possuem oito núcleos chamados Processadores Escalares (*Scalar Processors – SP*), duas unidades para números transcendentais (ex: π, e, etc.), uma unidade para controle das *threads* e a memória compartilhada. Os multiprocessadores criam, gerenciam e executam as *threads*, em hardware, e sem custo. Além disso, a barreira de sincronismo entre as *threads* é implementada em uma única instrução e resolvida de maneira muito eficiente (NVIDIA, 2009).

Para conseguir gerenciar centenas ou milhares de *threads* ao mesmo tempo, os multiprocessadores empregam a nova arquitetura chamada SIMT (Uma Instrução Múltiplas *Threads – Single-Instruction Multiple-Thread*). Nela cada *thread* é alocada a uma SP, que a executa independentemente. A unidade SIMT do multiprocessador cria, gerencia, agenda e executa as *threads* em grupos chamados *warps*. Os integrantes desses grupos são inicializados ao mesmo tempo, mas possuem liberdade para parar ou executar de forma independente.

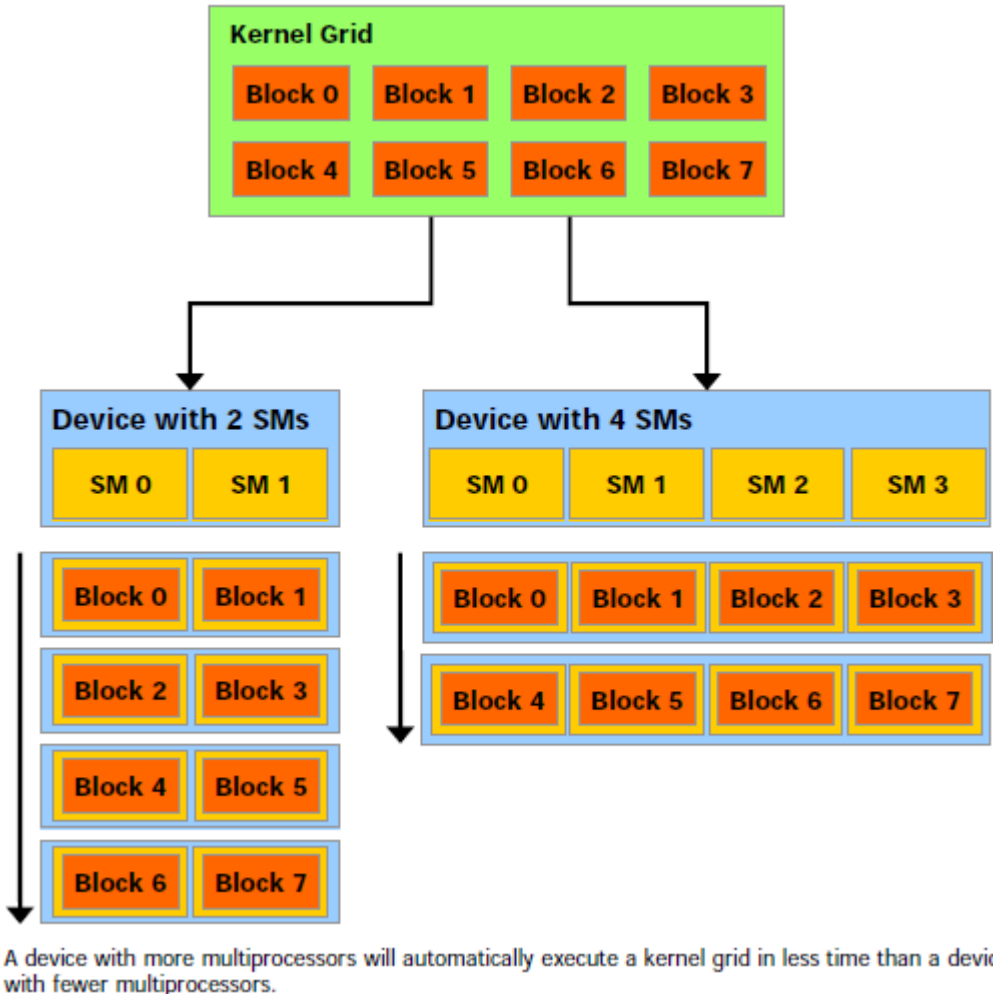
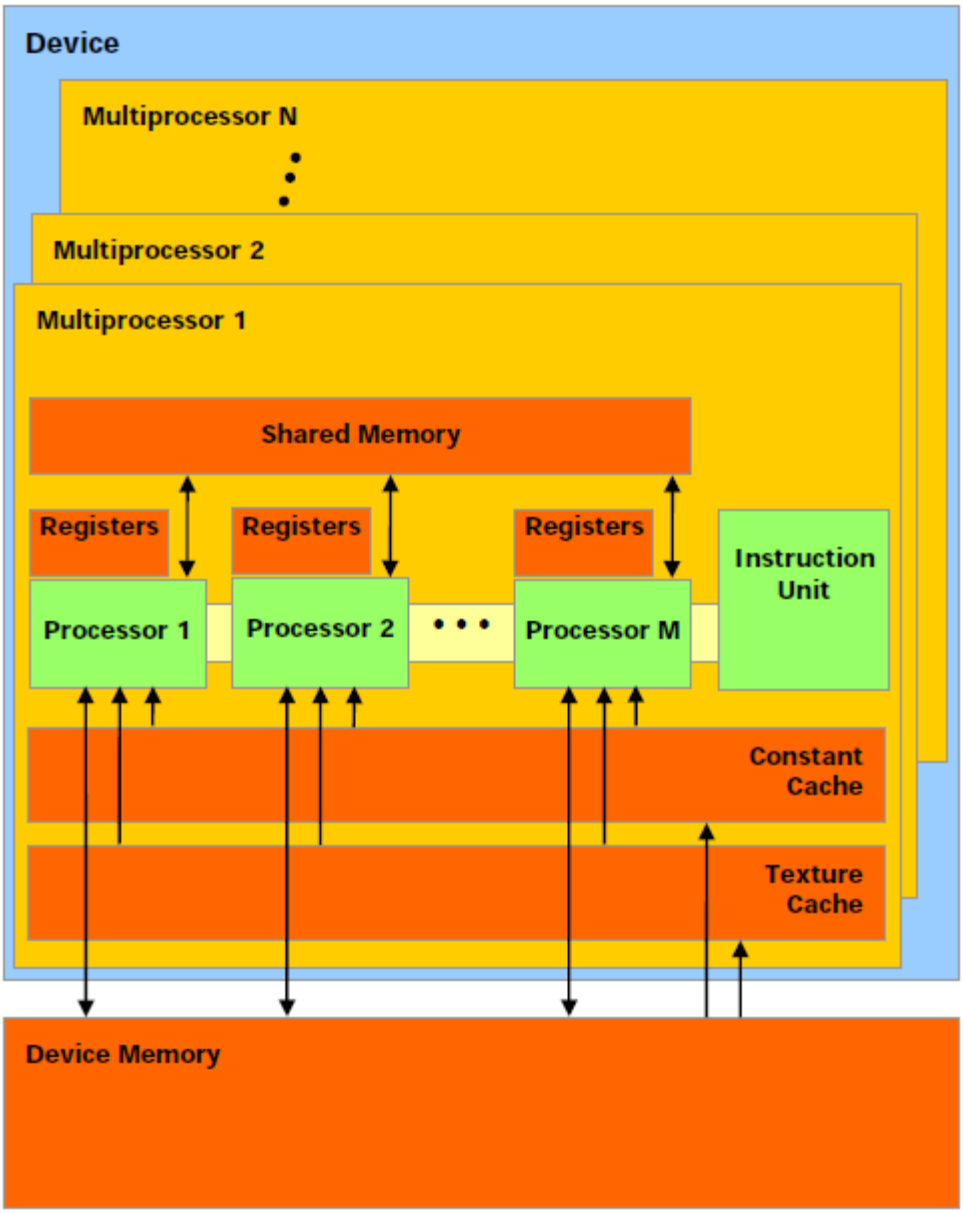



Figura 16 – Escalonamento automático feito em CUDA (NVIDIA, 2009).

A arquitetura SIMT é parecida com a arquitetura SIMD, onde um vetor de dados é processado pela mesma instrução, mas com uma diferença muito importante: na arquitetura SIMD, o tamanho do vetor é conhecido pelo hardware, enquanto que na arquitetura SIMT, as instruções especificam a execução e o comportamento de uma única *thread*.

Para concluir esta seção, a Figura 17 ilustra a estrutura da placa com os multiprocessadores e seus componentes, além dos diferentes tipos de memórias existentes.




A set of SIMT multiprocessors with on-chip shared memory.

Figura 17 – Modelo de hardware de uma placa gráfica com suporte a CUDA (NVIDIA, 2009).