I Escola Regional de Alto Desempenho de SP
ERAD-2010

MINI-CURSO:
Introdução à Programação em CUDA

Prof. Raphael Y. de Camargo
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC (UFABC)
raphael.camargo@ufabc.edu.br

Parte I

Introdução às GPUs

Placas Gráficas

Jogos 3D evoluíram muito nos últimos anos e passaram a exigir um poder computacional gigantesco

GPUs

Jogos modernos:

Além de gerar o cenário 3D, é preciso aplicar texturas, iluminação, sombras, reflexões, etc.

As folhas individuais das plantas são desenhadas

Sombras são calculadas dinamicamente

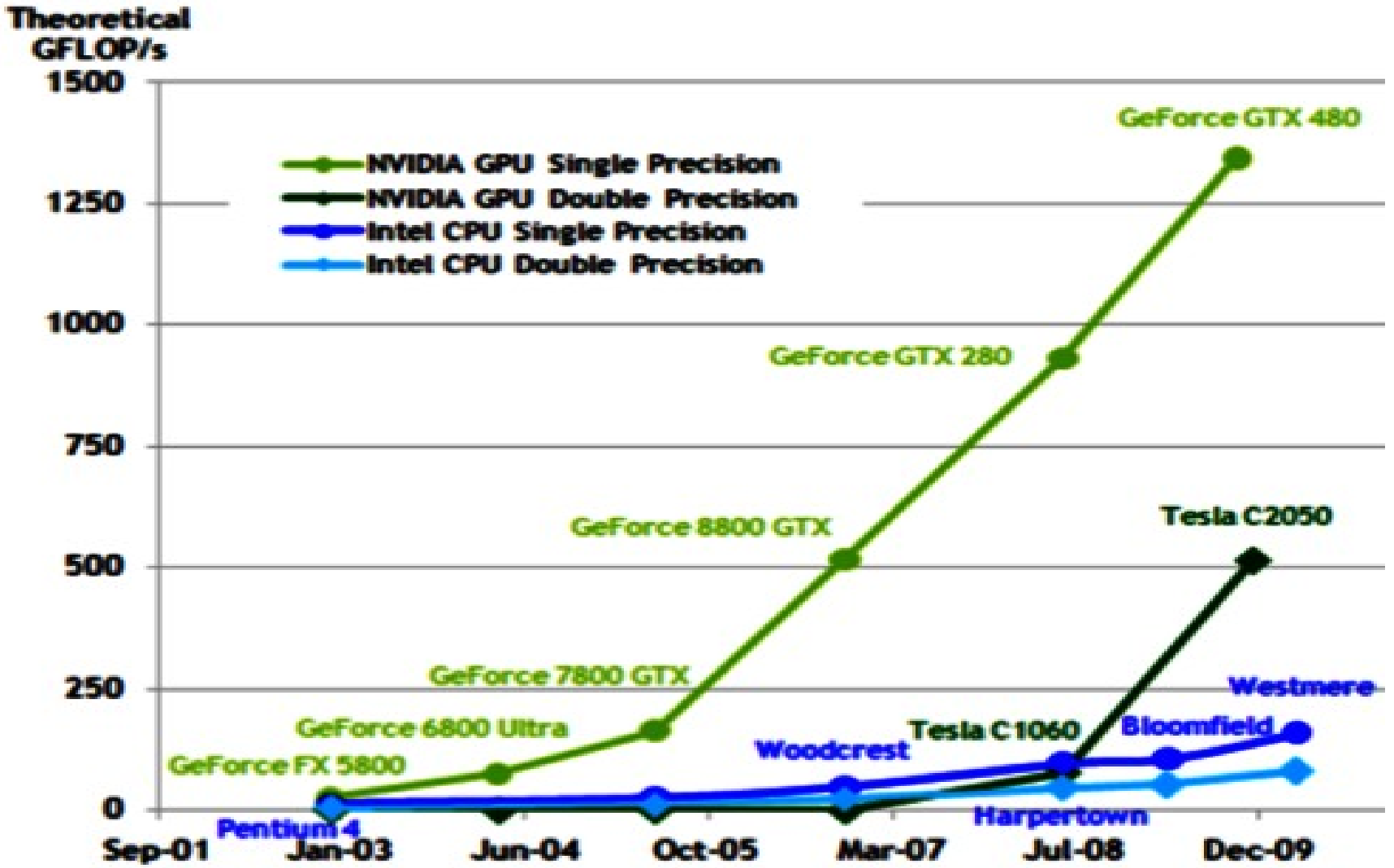
Para tal, as placas gráficas passaram a ser cada vez mais flexíveis e poderosas

NVIDIA GTX480: *480 núcleos viipreço US$ 500*

Desempenho máximo: 0.5 TFLOP (double) e 1.3 TFLOP (float)

Intel Core i7 980X: 6 núcleos viipreço US$ 1000

Desempenho máximo: 0.1 TFLOP (double)

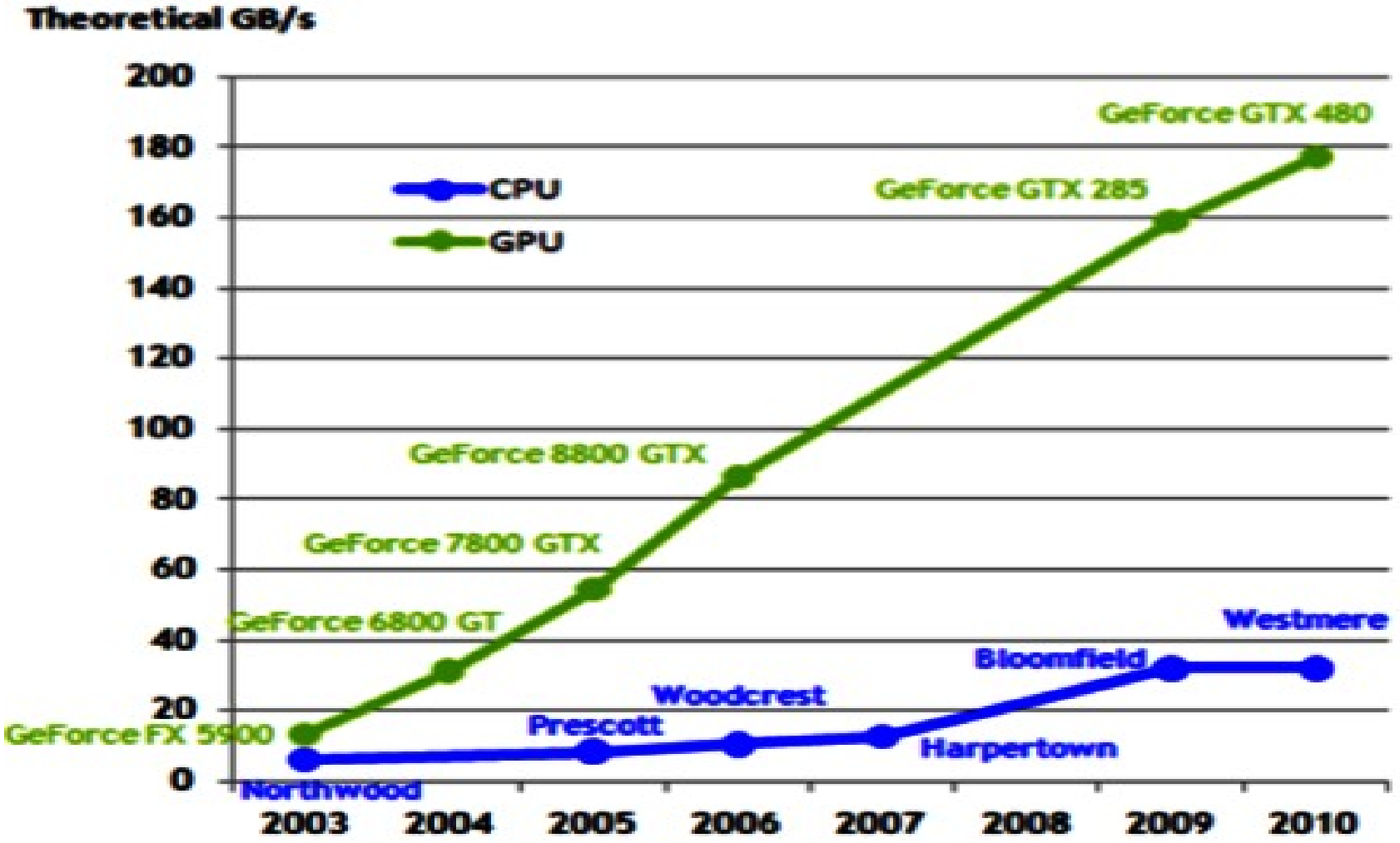


NVIDIA GTX480: *480 núcleos viipreço US$ 500*

1.5GB de Memória com acesso a 177 GB/s

Intel Core i7 980X: 6 núcleos viipreço US$ 1000

Acesso à memória a 25.6 GB/s



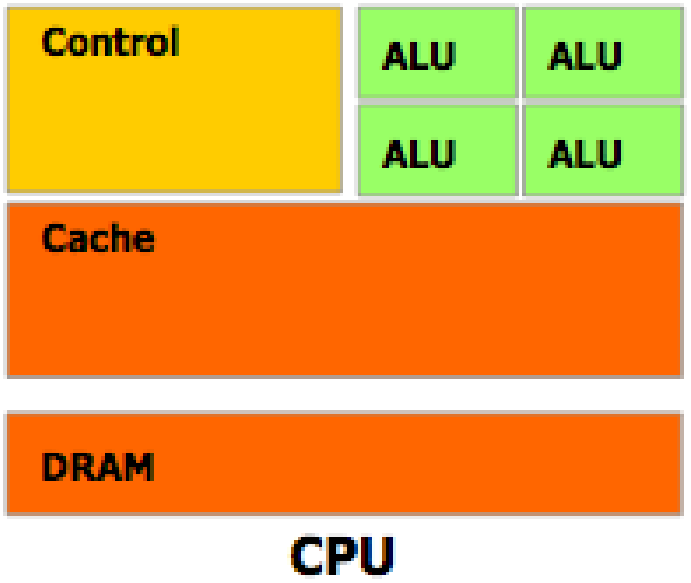
Arquitetura da GPU

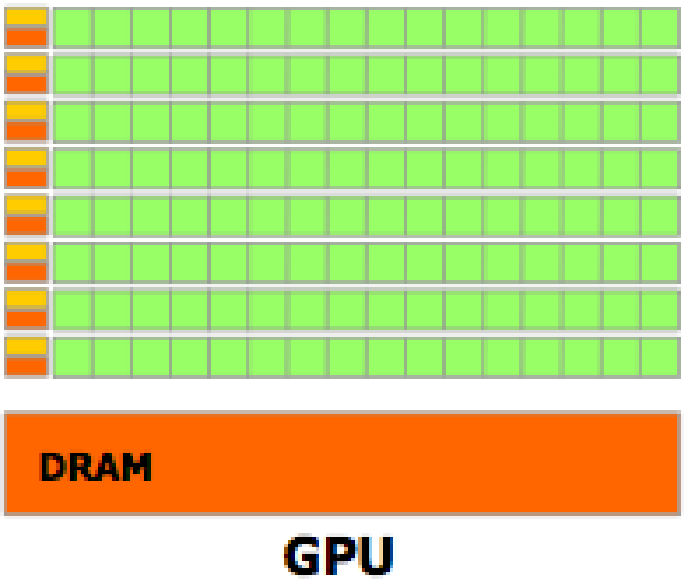
GPUs utilizam um maior número de transistores para colocar mais ALUs (Arithmetic Logic Unit) simplificadas

Permite fazer um maior número de cálculos ao mesmo tempo

Controle de fluxo mais simples:

- Feita para aplicações paralelas onde as mesmas operações são aplicadas sobre um grande conjunto de dados

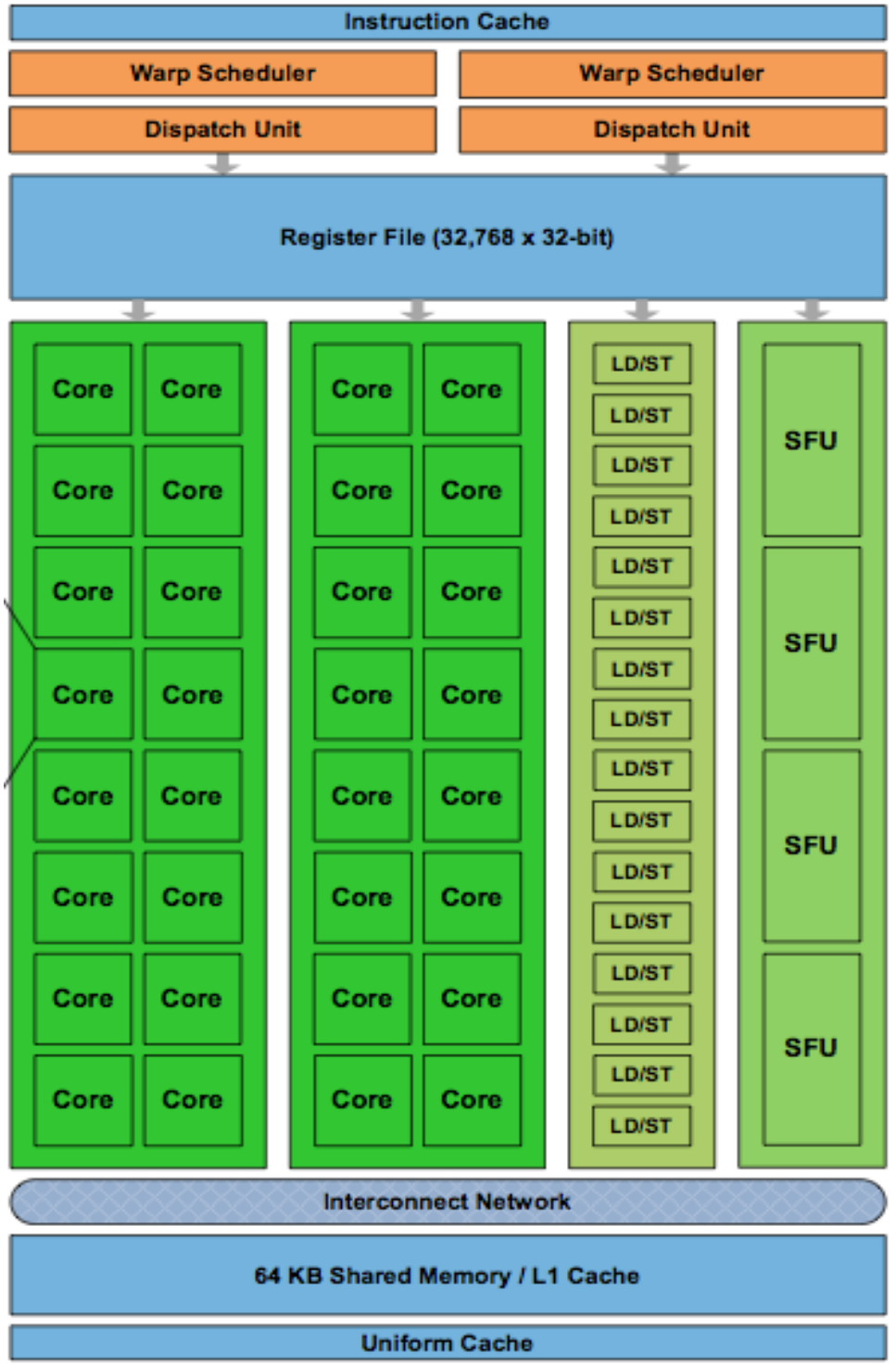




Arquitetura da GPU Fermi

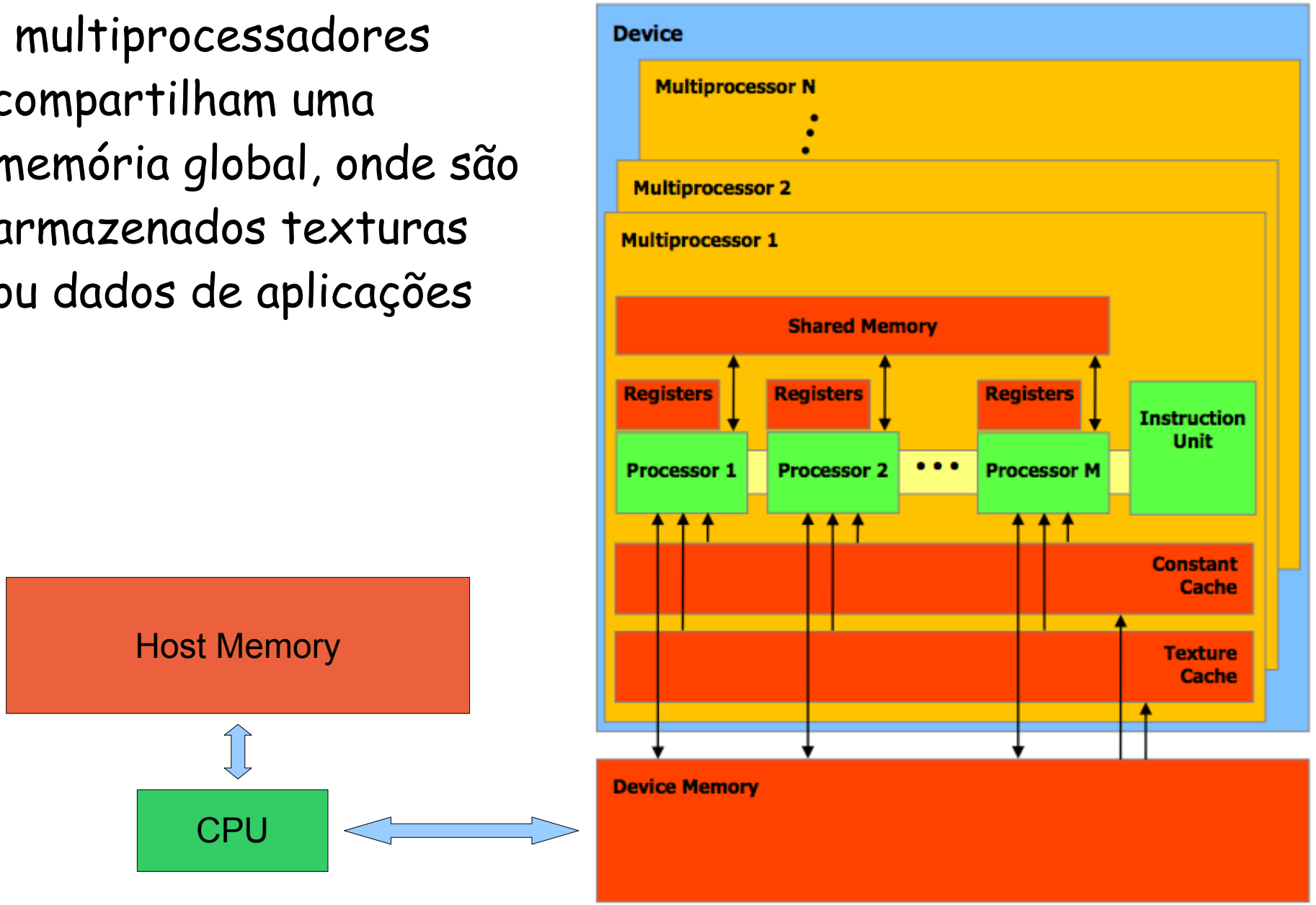
16 Multiprocessadores:

- 32 processadores
- 16 unidades Load/Store
- 4 unidades funções especiais
- 64 kB Memória compartilhada
- 32768 registradores
- Cache de constantes e textura



Arquitetura da GPU

Os multiprocessadores compartilham uma memória global, onde são armazenados texturas ou dados de aplicações



TESLA S2070

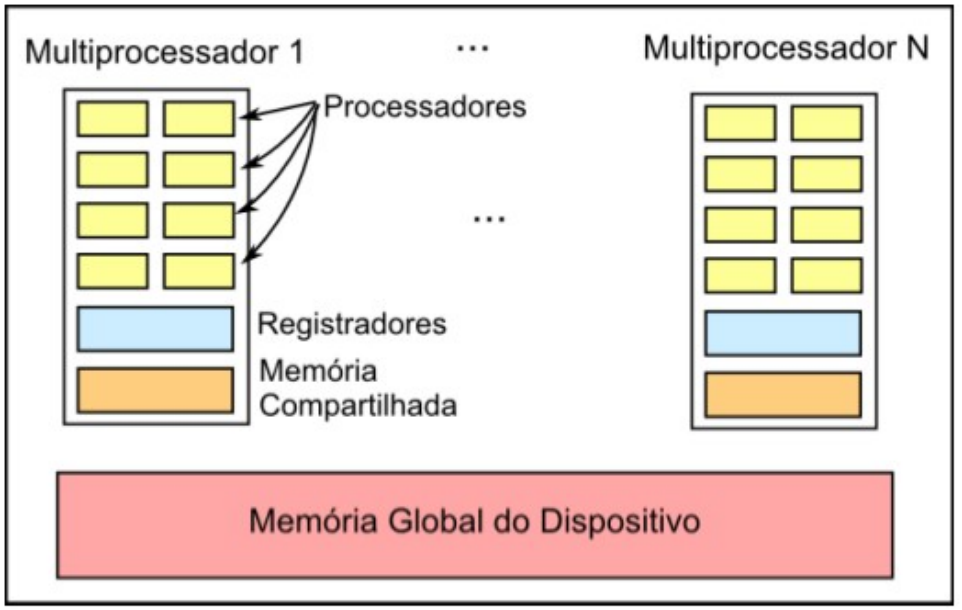
4 x 448 processadores

Form Factor	1U
# of Tesla GPUs	4
GPU Memory Speed	1.55 GHz
GPU Memory Interface	384-bit
GPU Memory Bandwidth	148 GB/sec
Double Precision floating point performance (peak)	2. Tflops
Single Precision floating point performance (peak)	4.13 Tflops
Total Dedicated Memory*	12GB GDDR5

O que é CUDA

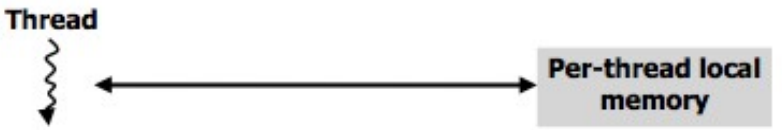
É uma arquitetura paralela de propósito geral destinada a utilizar o poder computacional de GPUs nVidia

Extensão da linguagem C, e permite controlar a execução de threads na GPU e gerenciar sua memória

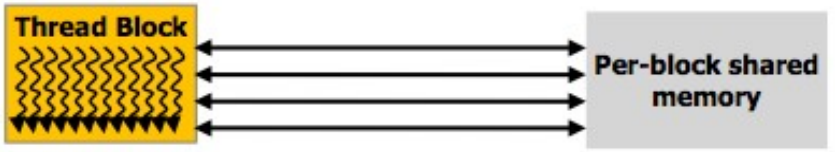


Arquitetura de uma GPU nVidia

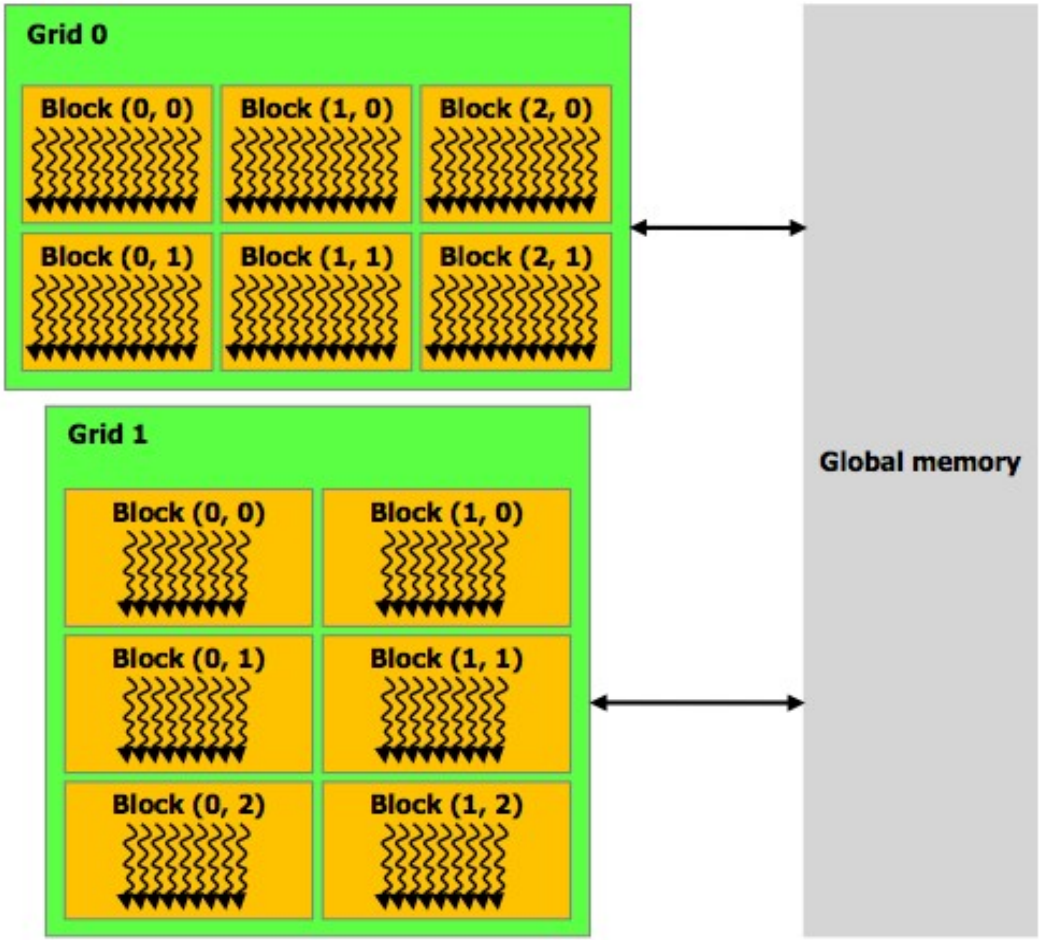
Hierarquia de Memória



Cada execução do kernel é composta por:



Grade → blocos → threads



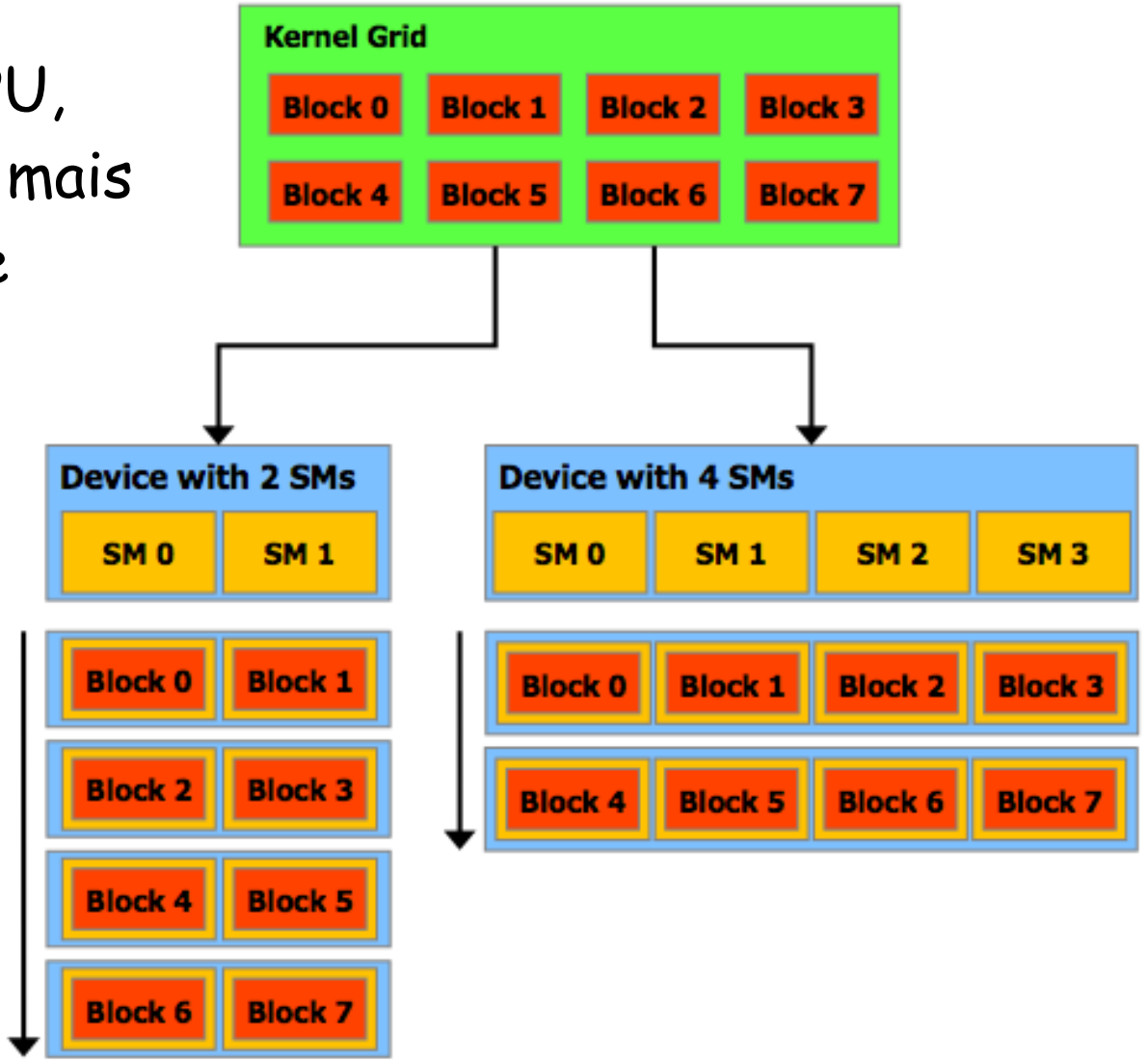
Hierarquia de memória:

- Registradores por thread
- Memória compartilhada por bloco
- Memória Global acessível a todas as threads

Execução de Aplicações

Cada bloco é alocado a um multiprocessador da GPU, que pode executar 1 ou mais blocos simultaneamente

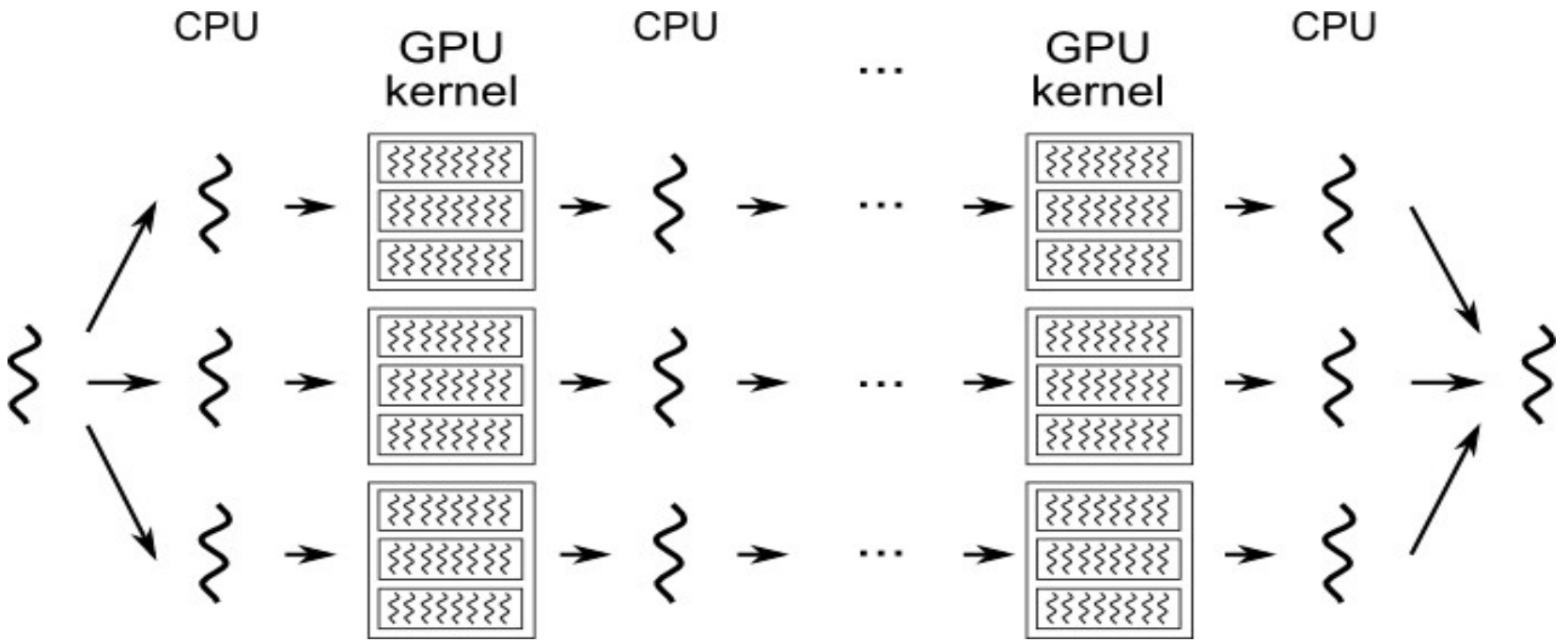
Cada multiprocessador executa 16 threads no modelo SIMT: Single Intruction, Multiple Thread



Modelo de Execução

Execução do programa controlada pela CPU que pode lançar *kernels*, que são trechos de código executados em paralelo por múltiplas threads na GPU

A execução de programas CUDA é composta por ciclos CPU, GPU, CPU, GPU, … , CPU, GPU, CPU.



Aplicações que rodam bem em GPUs

Programas que conseguem bons *speedups* em GPUs:

Podem ser subdivido em pequenos subproblemas, que são alocados a diferentes blocos e threads

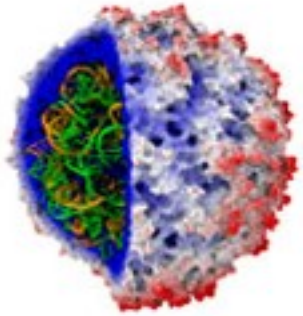
Cada thread mantém uma pequena quantidade de estado

Alta razão (operações de ponto flutuante) / (memória)

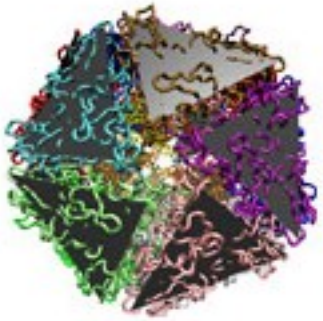
Os subproblemas são fracamente acoplados

Acoplamento: quando um problema pode ser dividido em subproblemas menores, o acoplamento indica o quanto os subproblemas são dependentes entre si.

CUDA for Research



Protective protein shell enclosing the viral genome

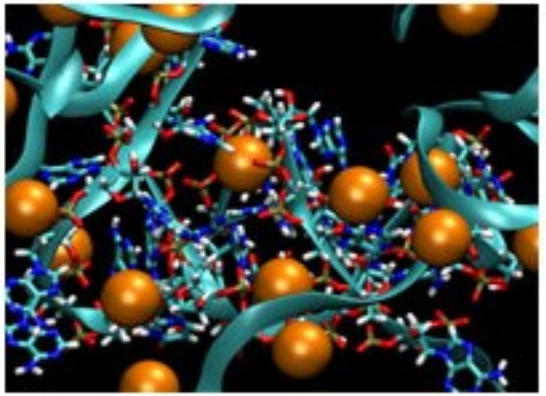


Icosahedral layout of the protein shell

Viruses, the cause of many diseases, are the smallest natural organisms known. Because of their simplicity and small size, computational biologists selected a virus for their first attempt to simulate an entire life form using a computer, choosing one of the tiniest, the satellite tobacco mosaic virus. Researchers simulated the virus in a drop of salt water using a program called NAMD (Nanoscale Molecular Dynamics) from the University of Illinois at Urbana-Champaign.

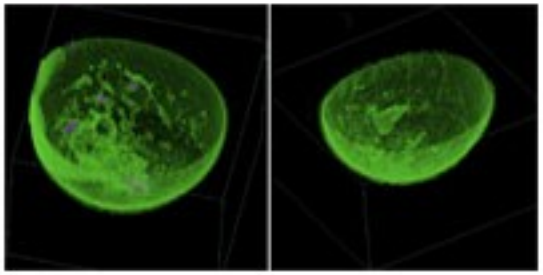
NAMD applications have been accelerated with CUDA, achieving dramatic speedups when running on a GPU-accelerated cluster at the National Center for Supercomputing Applications (NCSA). Researchers believe this step will assist modern medicine in efforts to better understand and treat viral illnesses.

For more information, please visit: www.ks.uiuc.edu/Research/namd/



Retirado de http://www.nvidia.com/cuda

CUDA for Medical



Volumetric rendering from TechniScan Whole Breast Ultrasound system

The ability to quickly produce highly-detailed images in a short timeframe is particularly relevant in the field of breast cancer scanning. Techniscan, a developer of automated ultrasound imaging systems, ported its proprietary algorithm from a traditional CPU-based system to CUDA and NVIDIA Tesla GPUs.

Retirado de http://www.nvidia.com/cuda

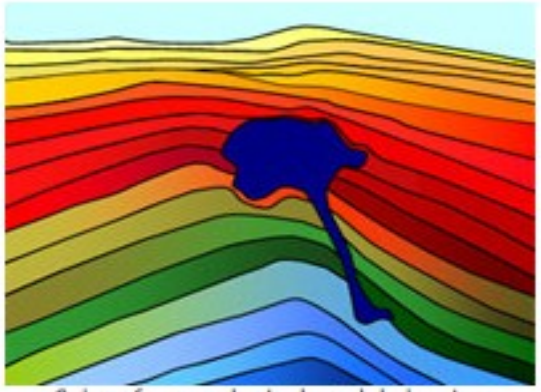
CUDA for Video and Photos

Choose a favorite video

As the popularity of digital media devices grows, home users are becoming increasingly frustrated by the time-consuming task of putting video onto these devices. Converting a two-hour movie, for instance, can take six or more hours when using the computer's CPU. Elemental's Badaboom is a video transcoding program that converts standard video files into formats that will run on the iPod and other portable devices.

Retirado de http://www.nvidia.com/cuda

CUDA for Energy

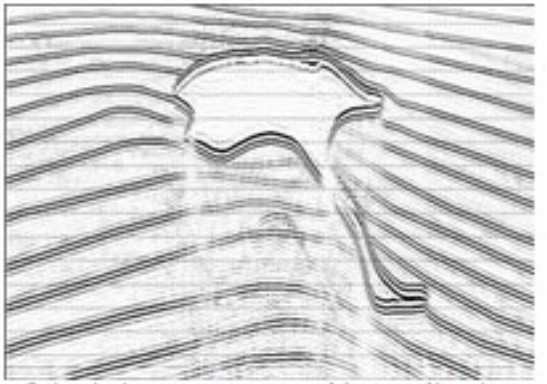


Subsurface geological model showing exploration targets below the salt body

The cost of drilling deep oil exploration wells can reach hundreds of millions of dollars. In many cases, there is only one chance to drill a successful well. SeismicCity develops and implements depth imaging technology used for interpretation of seismic data that leads to the selection of new drilling locations.

To improve the quality and effectiveness of their imaging, SeismicCity turned to CUDA and NVIDIA Tesla 8-series GPUs, achieving up to a 14X performance increase over a previous CPU-based configuration. With CUDA, SeismicCity is able to offer its clients commercial use of the most advanced depth imaging algorithms and dramatically improve the chances of successful drilling.

For more information, please visit: www.seismiccity.com.



Seismic image constructed by application of depth imaging technology

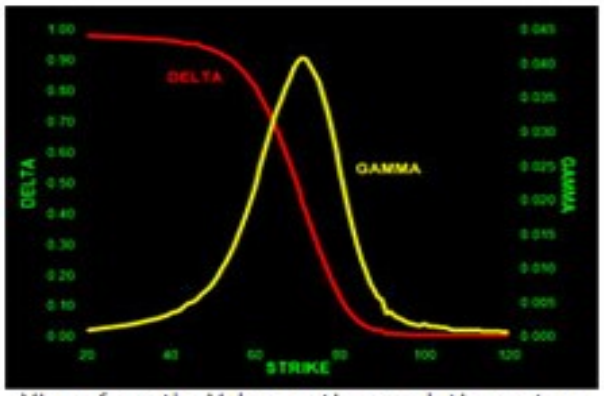
Retirado de http://www.nvidia.com/cuda

CUDA for Finance

Volera uses the GPU for real-time options analytics

For Hanweck Associates, a financial-services firm specializing in investment and risk management, it is imperative to offer clients a way to recalculate options in real time. Hanweck does this through its Volera line of high-performance options analytics. Using just 12 CUDA-enabled GPUs, Volera analyzes the entire U.S. equity options market in real time, a task that previously took more than 60 conventional servers. With CUDA, Hanweck's clients can achieve faster visibility into today's high-speed markets while realizing substantial savings on power consumption, hardware costs, and data center real estate.

For more information, please visit www.hanweckassoc.com.



Views from the Volera options analytics system

Views from the Volera options analytics system

Retirado de http://www.nvidia.com/cuda

GPUs vs Supercomputadores

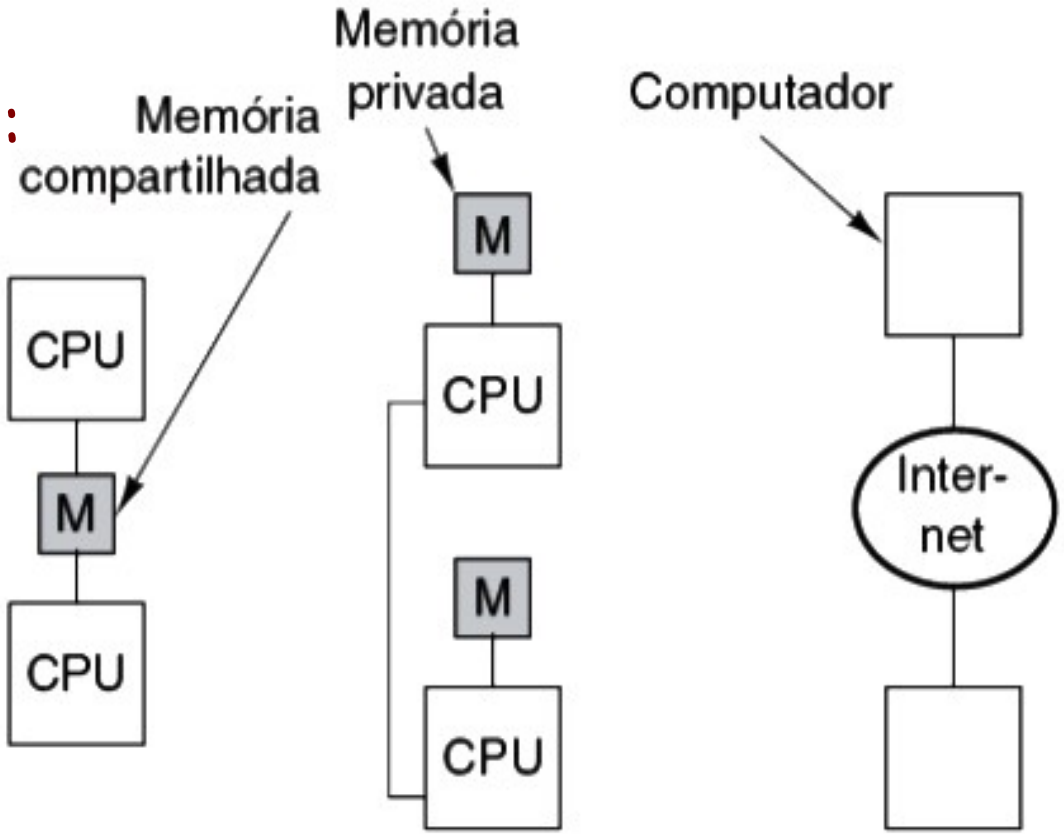
Sistemas de multiprocessadores: Conjunto de unidades de processamento e memória conectados por uma rede de interconexão

Sistemas classificados como:

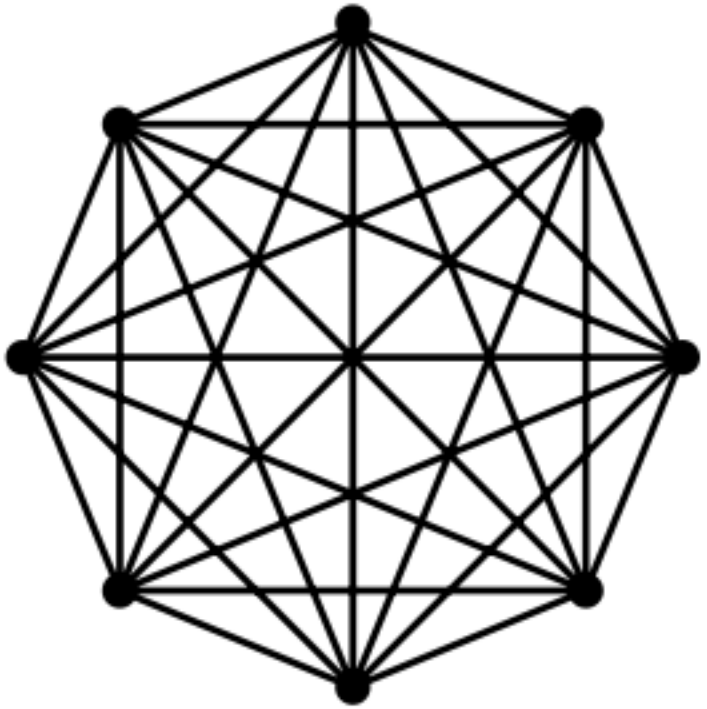
- Memória compartilhada
- Troca de mensagens

Exemplos:

- Supercomputadores
- Aglomerados (clusters)

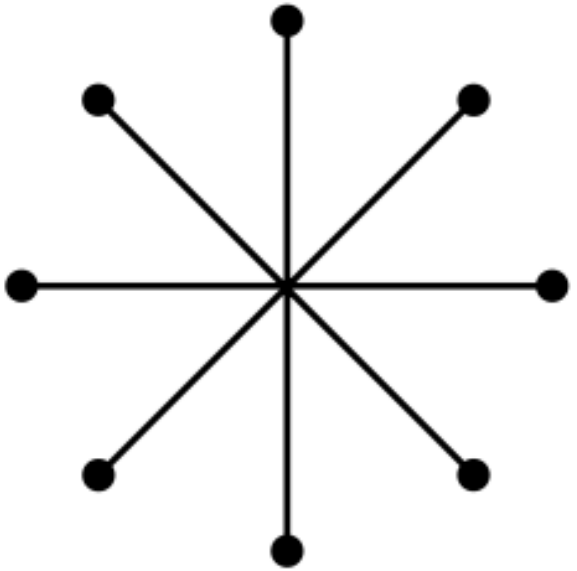


Exemplos de Redes Estáticas



Alta tolerância a falhas

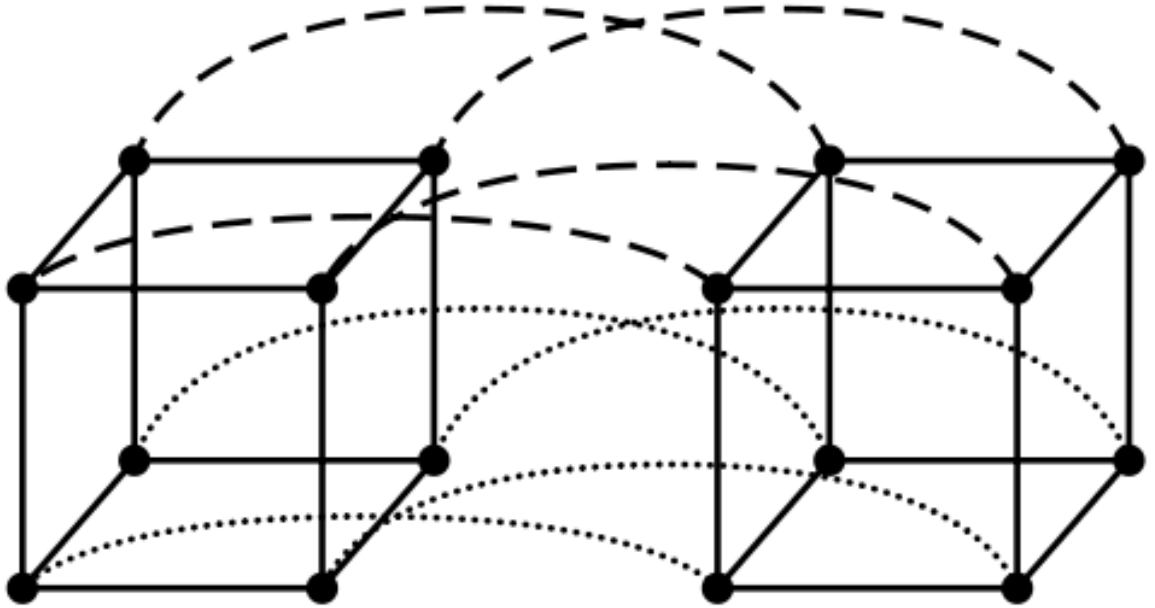
No de conexões: $O(N^2)$

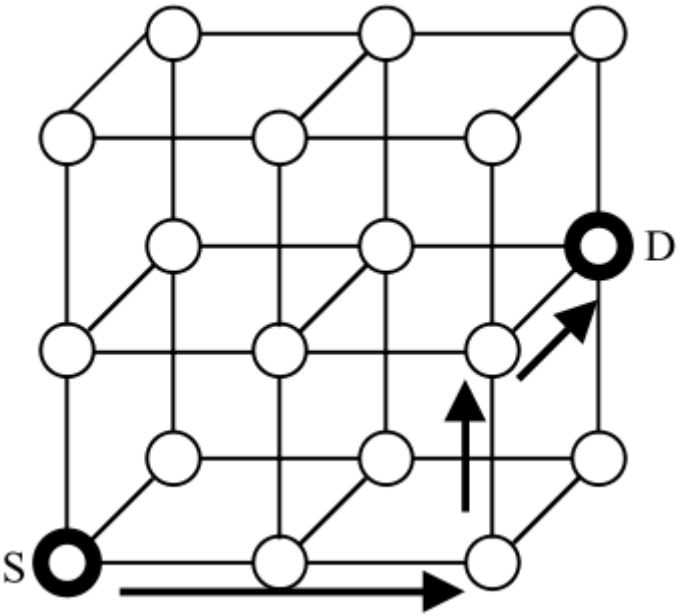


Ponto único de falha e gargalo

No de conexões: $O(N)$

Exemplos de Redes Estáticas





k-cubo: nós: O (2^k),
Conexões: O (2^k*k/2)

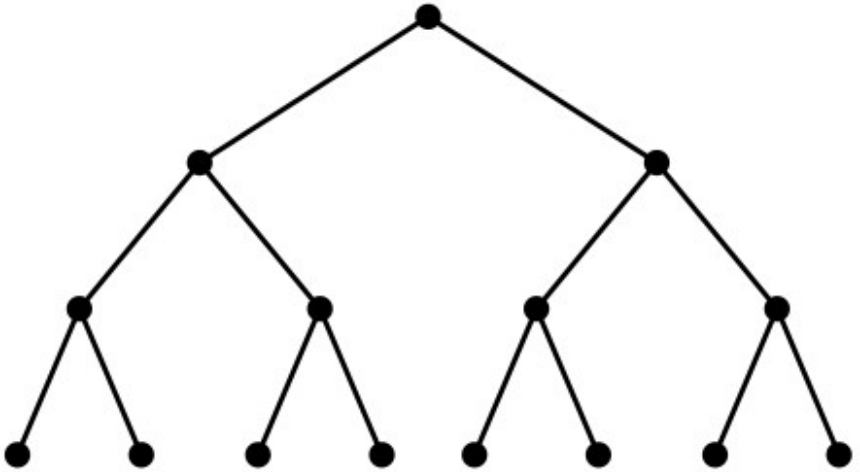
Hypercubo 10-dim

1024 nós e 5120 conexões

Malha com k dimensões
No de conexões: O(N*k/2)

Mais fácil de construir

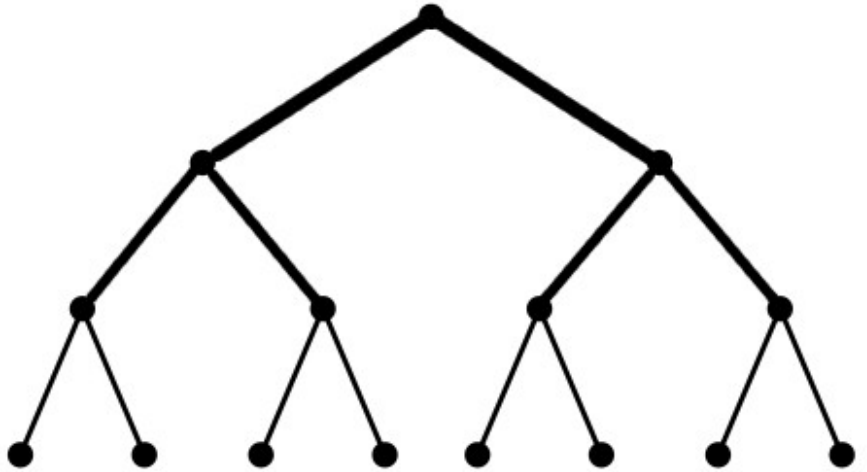
Exemplos de Redes Estáticas



Árvore

Conexões: O(N)
Atraso: O(log N)

Nó raiz é gargalo e ponto único de falha

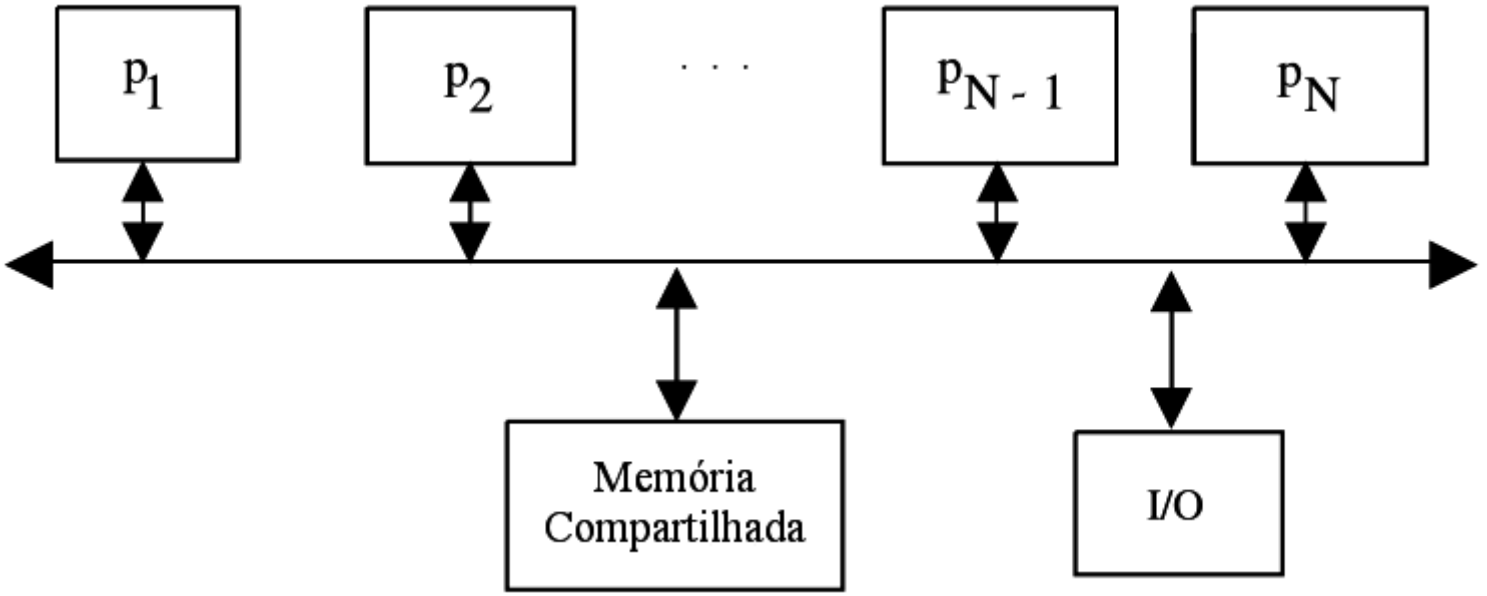


Árvore Gorda

Nós mais altos com maior largura de banda

Normalmente implementadas com múltiplas conexões

Rede Dinâmica: Barramento Simples

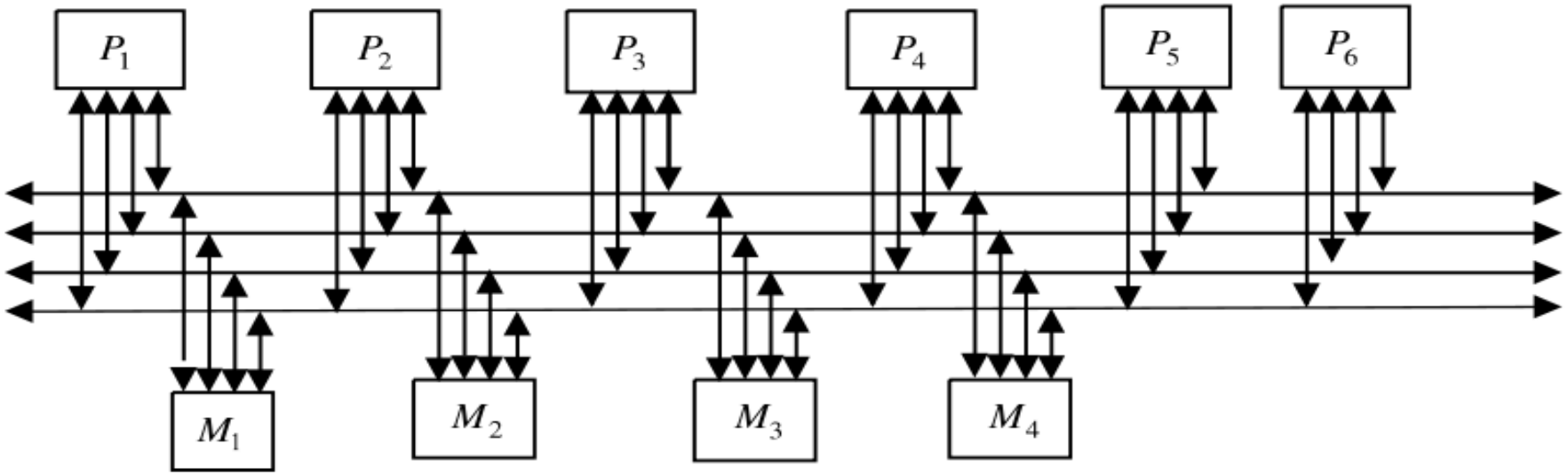


Todas os processadores ligados à memória por um barramento simples

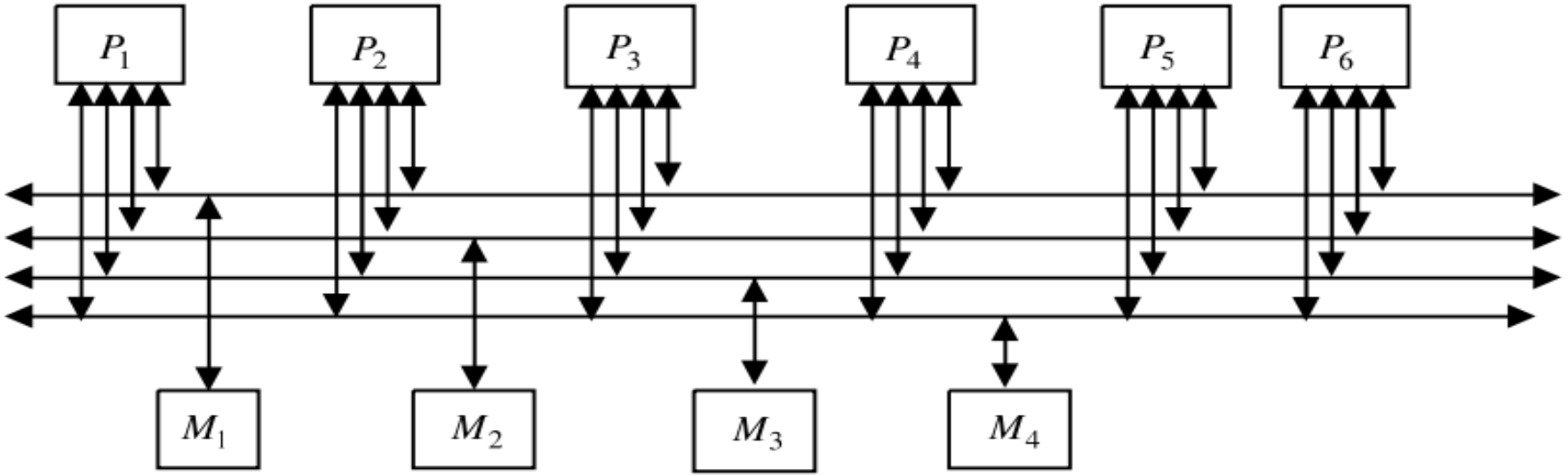
Fácil e barato de implementar

Mas o barramento rapidamente se torna um gargalo

Múltiplos Barramentos



(a)



Exemplo: SGI Altix 4700

- Fabricante: Silicon Graphics, Inc
- Memória: **272 GB**
- Processadores: 68 Dual-Core Itanium 9000 (64 bits)
- Sistemas Operacionais: Linux (SUSE) e Enterprise Server 10
- Capacidade de Disco: 30TB (InfiniteStorage 350 e 120)
- Custo: R$ 2 milhões

Fonte: http://www.ufabc.edu.br

Detalhes do Altix 4700

- Projeto modular
  - Placas blade com 2 sockets e 8 DIMM
  - Memória de até 128TB e até centenas de processadores
- Software
  - C/C++ e Fortran, Depuradores, ferramentas de análise
  - MPI, OpenMP, Ferramentas de threading
- Rede de Interconexão SGI NUMAlink 4
  - Topologia árvore gorda (fat tree)
  - 6.4GB/sec bidirectional bandwidth per link

Fonte: http://www.sgi.com

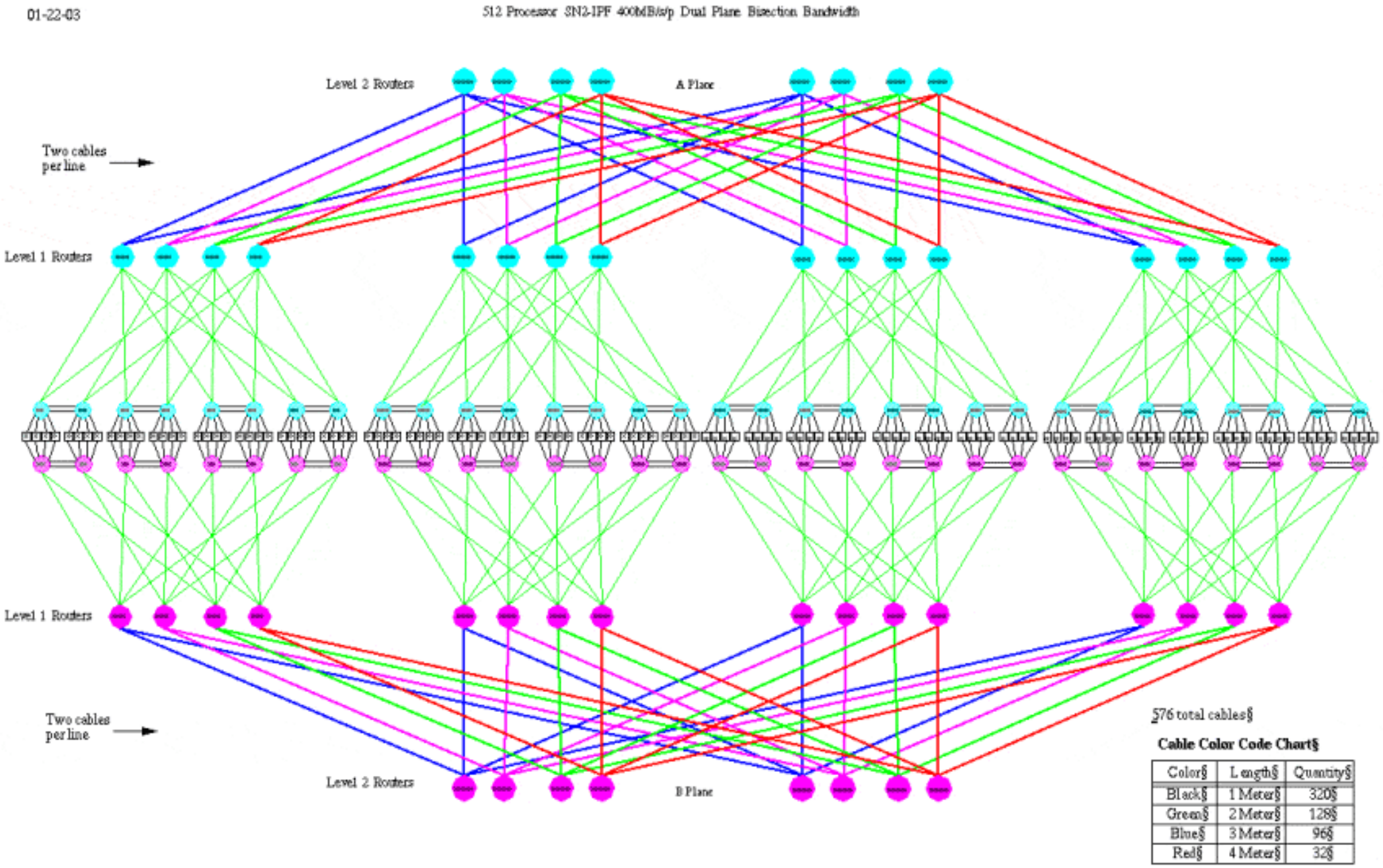


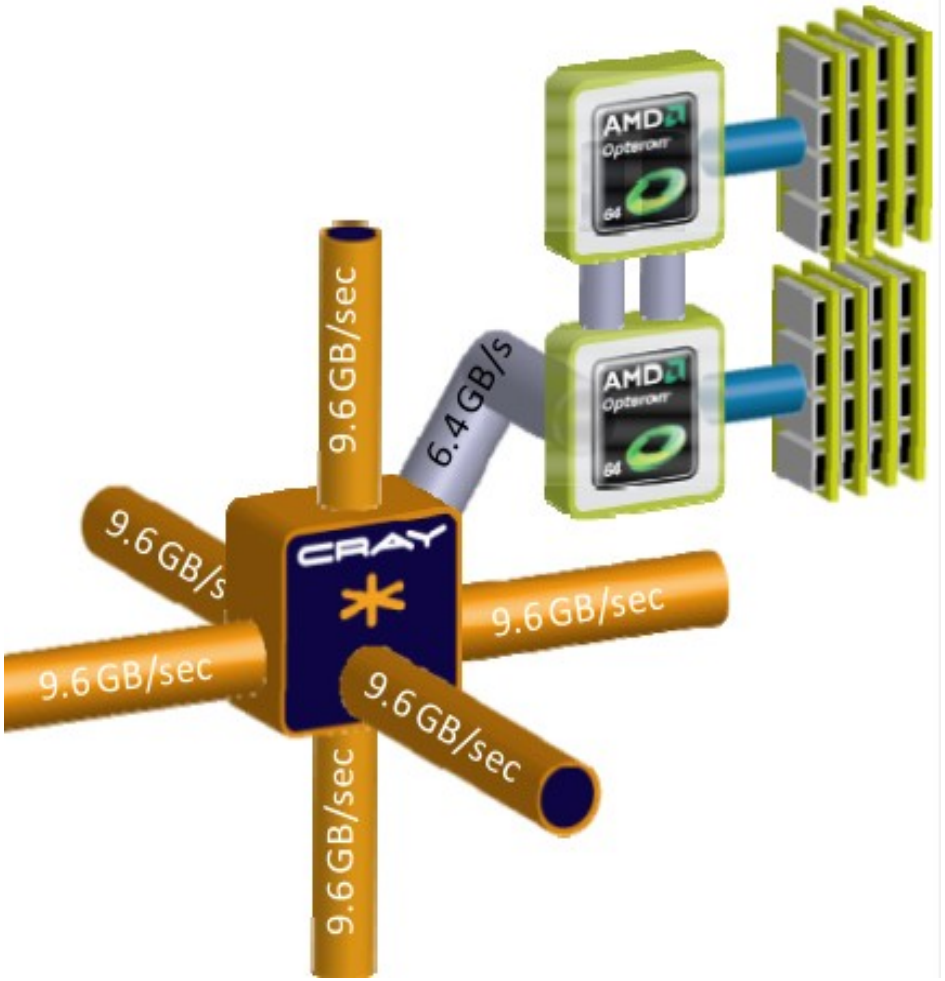
Figure 2. 512-Processor Dual "Fat-Tree" Interconnect Topology

Fonte: http://www.embedded-computing.com/articles/woodacre/

Supercomputador Moderno

- Jaguar: Oak Ridge National Laboratory

	XT5	XT4	Total
Cabinets	200	84	284
Compute Blades	4,672	1,958	6,630
Quad-core Opteron Processors	37,376	7,832	45,208
Cores	149,504	31,328	180,832
Peak TeraFLOPS	1,375	263	1,639
Nodes	18,688	7,832	26,520
Memory (TB)	300	62	362
Number of disks	13,440	2,774	16,214
Disk Capacity (TB)	10,000	750	10,750
I/O Bandwidth (GB/s)	240	44	284



Supercomputador Moderno

Blue Gene/L at LLNL, 131,072 processors

Extraído de http://www.ks.uiuc.edu/Research/namd/motivation/

Aplicações para Supercomputadores

Qualquer tipo de aplicação

Quanto menor o acoplamento, maior o desempenho

Mas normalmente são aplicações com alto grau acoplamento

Justificam o alto preço da rede de interconexão

Exemplos:

- Previsão do tempo, evolução do clima
- Mecânica dos Fluidos (projeto de carros e aviões)
- Biologia Computacional: enovelamento de proteínas, redes de interação gênica, etc.

TESLA M2070

448 processadores

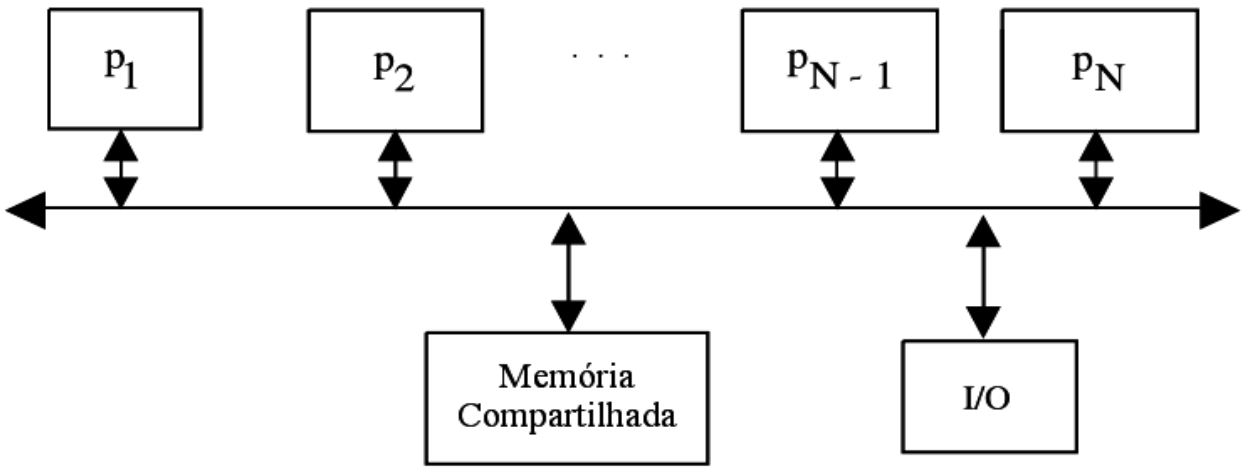
Form Factor	9.75" PCIe x16 form factor
# of Tesla GPUs	1
Double Precision floating point performance (peak)	515 Gflops
Single Precision floating point performance (peak)	1.03 Tflops
Total Dedicated Memory* Tesla M2050 Tesla M2070	 3GB GDDR5 6GB GDDR5
Memory Speed	1.55 GHz
Memory Interface	384-bit
Memory Bandwidth	148 GB/sec

TESLA S2070

4 x 448 processadores

Form Factor	1U
# of Tesla GPUs	4
GPU Memory Speed	1.55 GHz
GPU Memory Interface	384-bit
GPU Memory Bandwidth	148 GB/sec
Double Precision floating point performance (peak)	2. Tflops
Single Precision floating point performance (peak)	4.13 Tflops
Total Dedicated Memory*	12GB GDDR5

Barramento de GPUs



Barramento de memória compartilhado

Todos os processadores acessam a memória global utilizando o mesmo barramento

Mecanismos primitivos de sincronização entre threads

Acesso à memória do computador: PCI-Express (8 GB/s)

Aplicações que rodam bem em GPUs

Programas que conseguem bons *speedups* em GPUs:

Podem ser subdivido em pequenos subproblemas, que são alocados a diferentes blocos e threads

Cada thread mantém uma pequena quantidade de estado

Alta razão (operações de ponto flutuante) / (memória)

Os subproblemas são fracamente acoplados

Exemplos: Algoritmos genéticos, codificação de vídeo, processamento de imagens médicas, etc.

Parte II

Introdução à Programação em CUDA

O que é CUDA

É uma arquitetura paralela de propósito geral destinada a utilizar o poder computacional de GPUs nVIDIA

Extensão da linguagem C, que permite o uso de GPUs:

- Suporte a uma hierarquia de grupos de threads
- Definição de kernels que são executados na GPU
- API com funções, que permitem o gerenciamento da memória da GPU e outros tipos de controle

Obtendo o CUDA

CUDA pode ser obtido gratuitamente no site da nVidia:

http://www.nvidia.com/object/cuda_get.html

Disponível para Windows (XP, Vista e 7), Linux e MacOS X, em versões de 32 e 64 bits. É composto por:

CUDA Driver: Permite o acesso ao hardware

CUDA Toolkit: Ferramentas e bibliotecas para programação em CUDA

CUDA SKD: Exemplos de código

Exemplo de Instalação do CUDA

Para instalar o toolkit, basta rodar o instalador:

```
> chmod +x cudatoolkit_2.3_linux_64_ubuntu9.04.run.sh
> ./cudatoolkit_2.3_linux_64_ubuntu9.04.run.sh
```

Digite os comandos para configurar a instalação:

```
> export PATH=$PATH:/home/raphael/cuda/bin
> export LD_LIBRARY_PATH=
        $LD_LIBRARY_PATH:/home/raphael/cuda/lib64
```

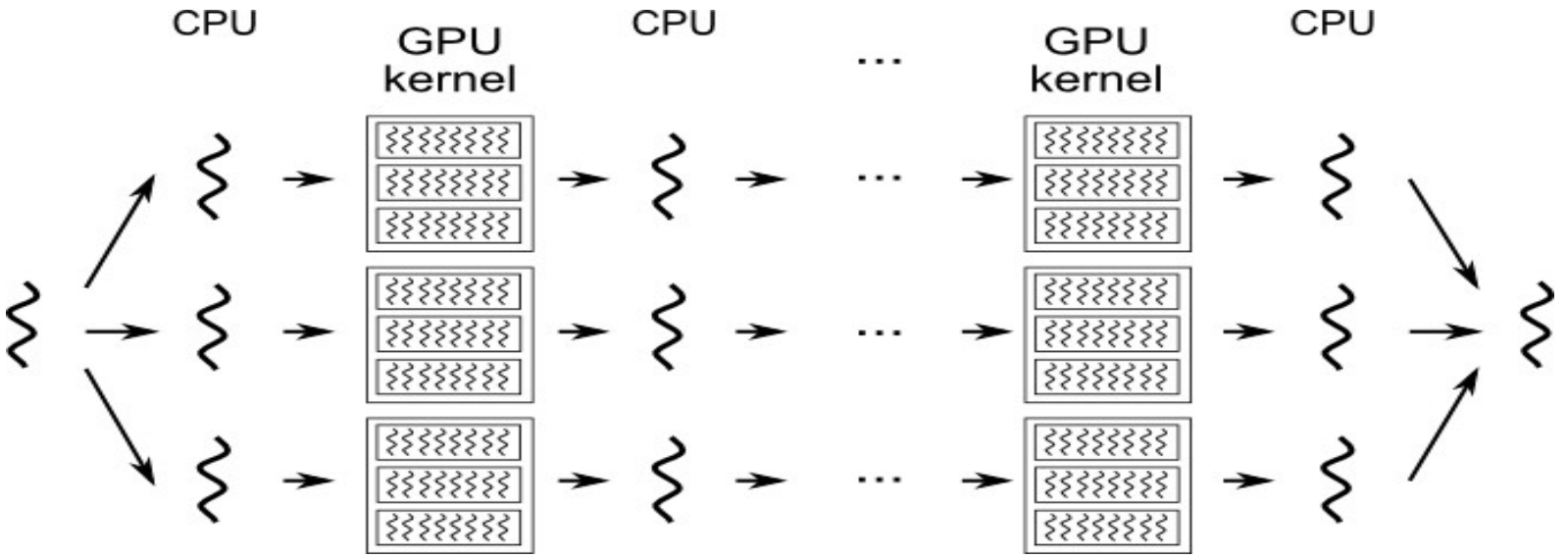
Agora o Toolkit está instalado e configurado!

Vamos para o primeiro exemplo.

Modelo de Execução

Execução do programa controlada pela CPU que pode lançar *kernels*, que são trechos de código executados em paralelo por múltiplas threads na GPU

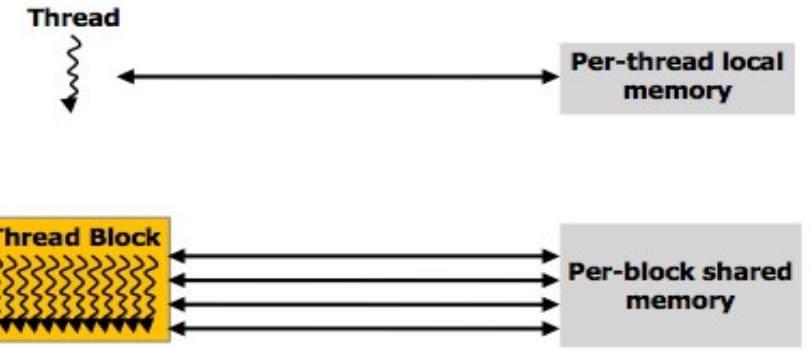
A execução de programas CUDA é composta por ciclos CPU, GPU, CPU, GPU, … , CPU, GPU, CPU.



Hierarquia de Memória

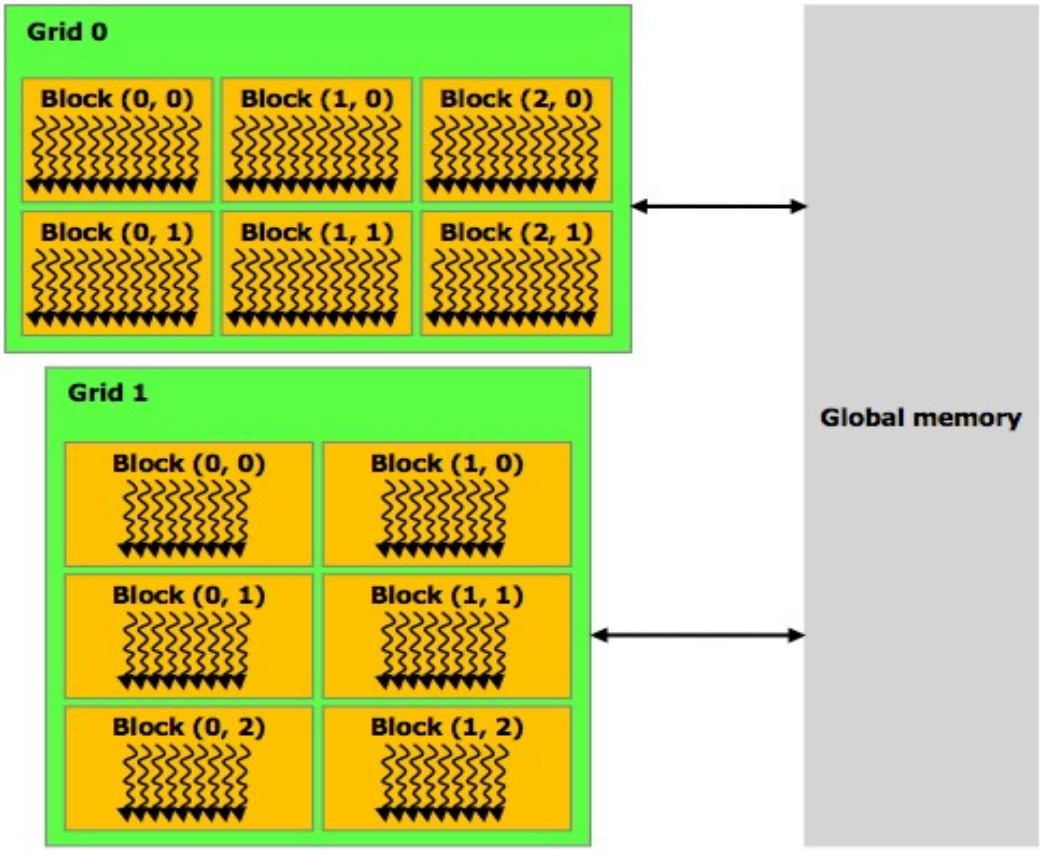
Cada execução do kernel é composta por:

Grade → blocos → threads



Hierarquia de memória:

- Registradores por thread
- Memória compartilhada por bloco
- Memória Global acessível a todas as threads



Exemplo Simples

Neste primeiro exemplo iremos aprender como criar um kernel simples, que realiza a soma de 2 vetores.

Veremos as principais operações usadas em CUDA:

Alocação e liberação de memória, transferência de dados, lançamento do kernel

O código abaixo contém erros e limitações que iremos resolver na aula.

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

```
// Host code
int main()  {
    int n = 5;
    size_t size = n * sizeof(float);
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    float h_A[] = {1,2,3,4,5};
    float h_B[] = {10,20,30,40,50};
    float h_C[] = {0,0,0,0,0};

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int nThreadsPerBlock  = 256;
    int nBlocks    = n / nThreadsPerBlock;

    VecAdd<<<nBlocks, nThreadsPerBlock>>>(d_A, d_B, d_C);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A);  cudaFree(d_B); cudaFree(d_C);
}
```

Avaliação do Exemplo I

Para compilar o código, basta utilizar o comando:

nvcc -o ex1 ex1.cu

Perguntas:

1) O programa funciona corretamente? Teste seu funcionamento imprimindo o resultado obtido

2) Corrija o programa

3) Aumente o tamanho dos vetores para, por exemplo, 1024. Teste o resultado e, se não for o esperado, corrija o programa.

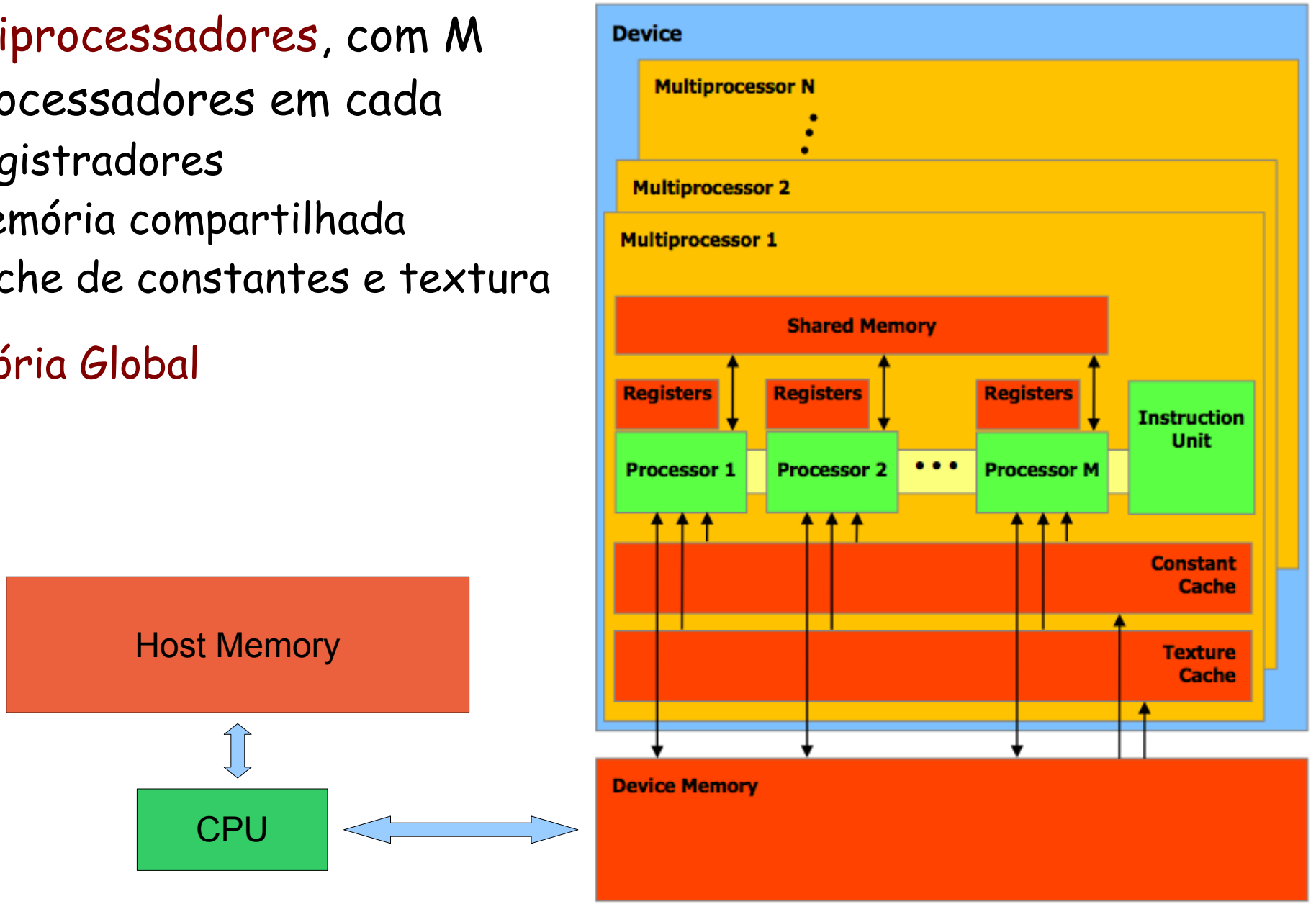
Obs: Você deve manter o número de threads por bloco em 256.

Dica: O valor de blockIdx.x devolve o bloco ao qual a thread pertence

Arquitetura da GPU

Multiprocessadores, com M processadores em cada
- Registradores
- Memória compartilhada
- Cache de constantes e textura

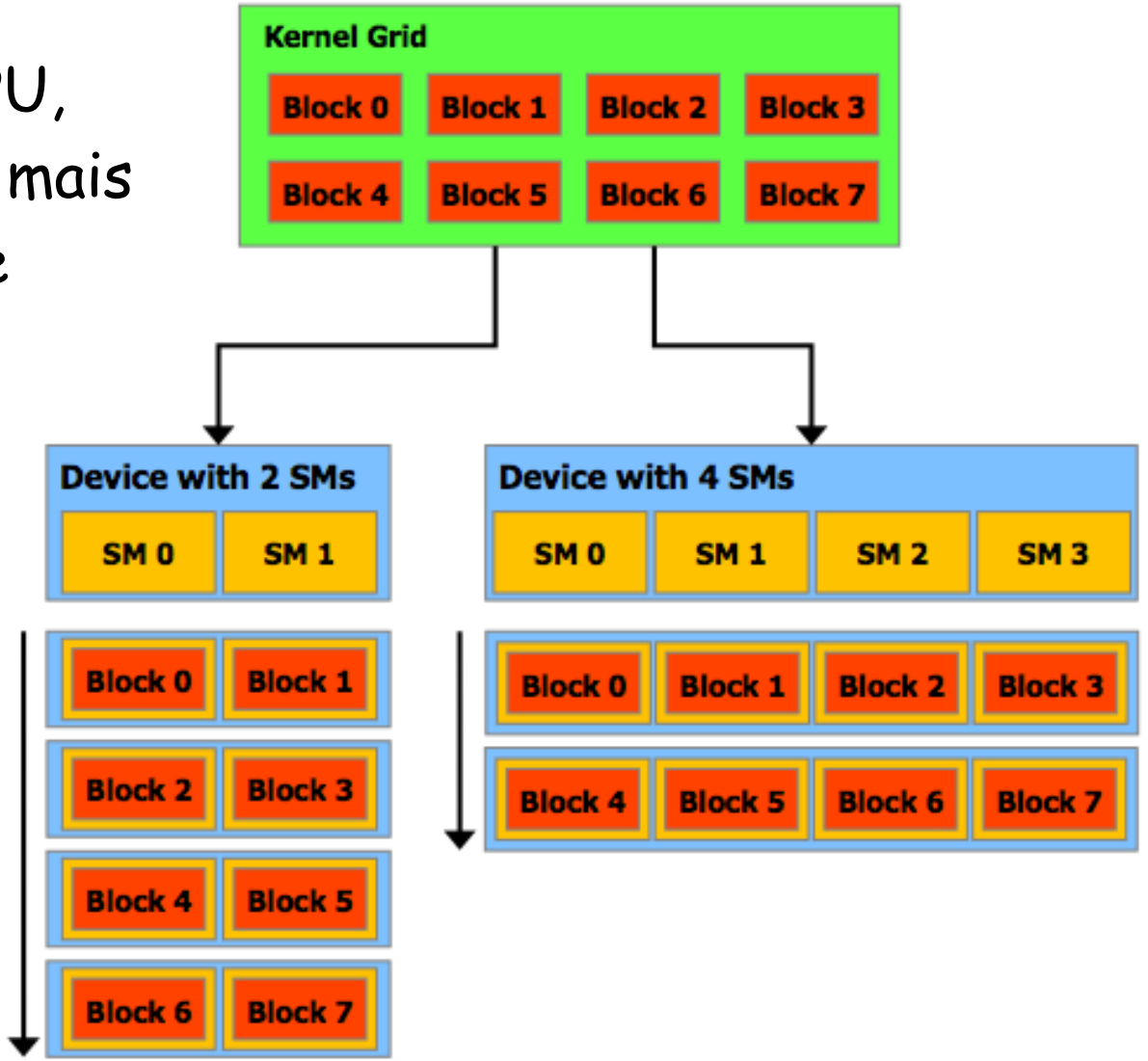
Memória Global



Execução de Aplicações

Cada bloco é alocado a um multiprocessador da GPU, que pode executar 1 ou mais blocos simultaneamente

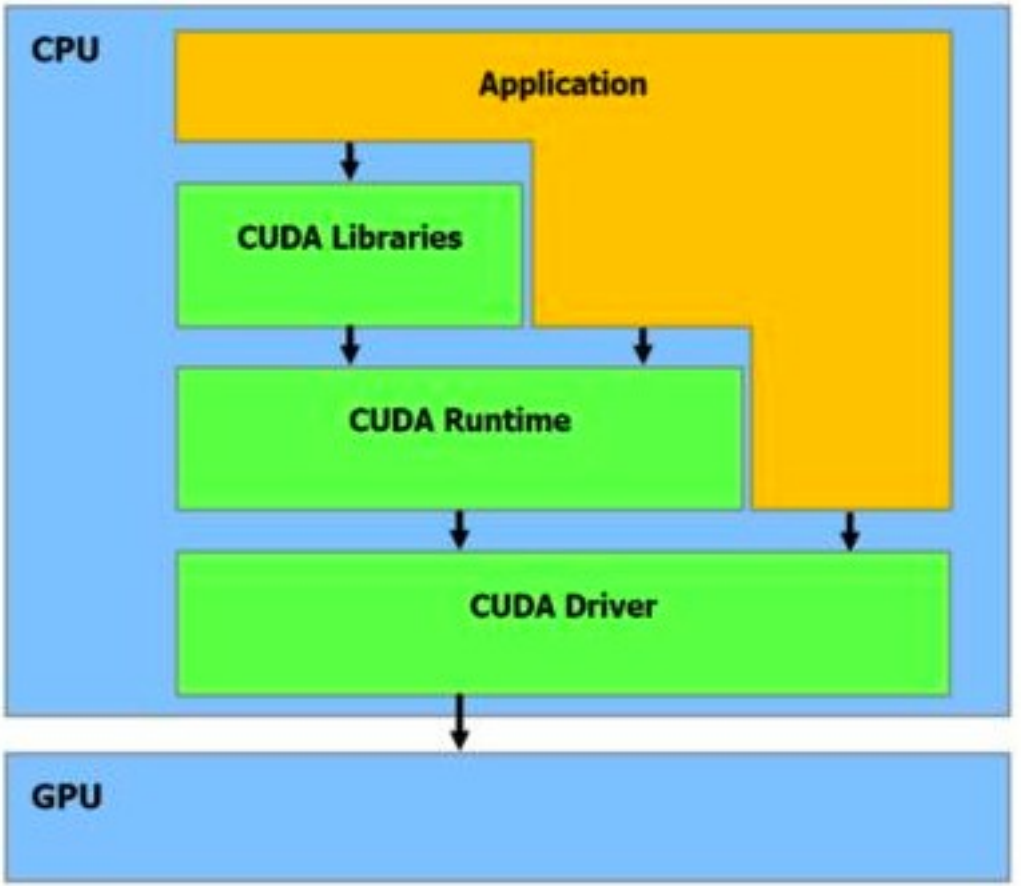
Cada multiprocessador executa 16 threads no modelo SIMT: Single Intruction, Multiple Thread



CUDA Runtime vs Driver API

Programas em CUDA normalmente utilizam o CUDA Runtime, que fornece primitivas e funções de alto-nível

Além disso, é possível utilizar também a API do driver, que permite um melhor controle da aplicação



CUDA 3.1 (Junho de 2010)

Suporte a múltiplos (16) kernels simultâneos

Melhorias no CUDA:

- Suporte a printf no código do dispositivo
- Suporte a ponteiros com endereços de funções
- Suporte a funções recursivas

Interoperabilidade entre API do driver e CUDA Runtime

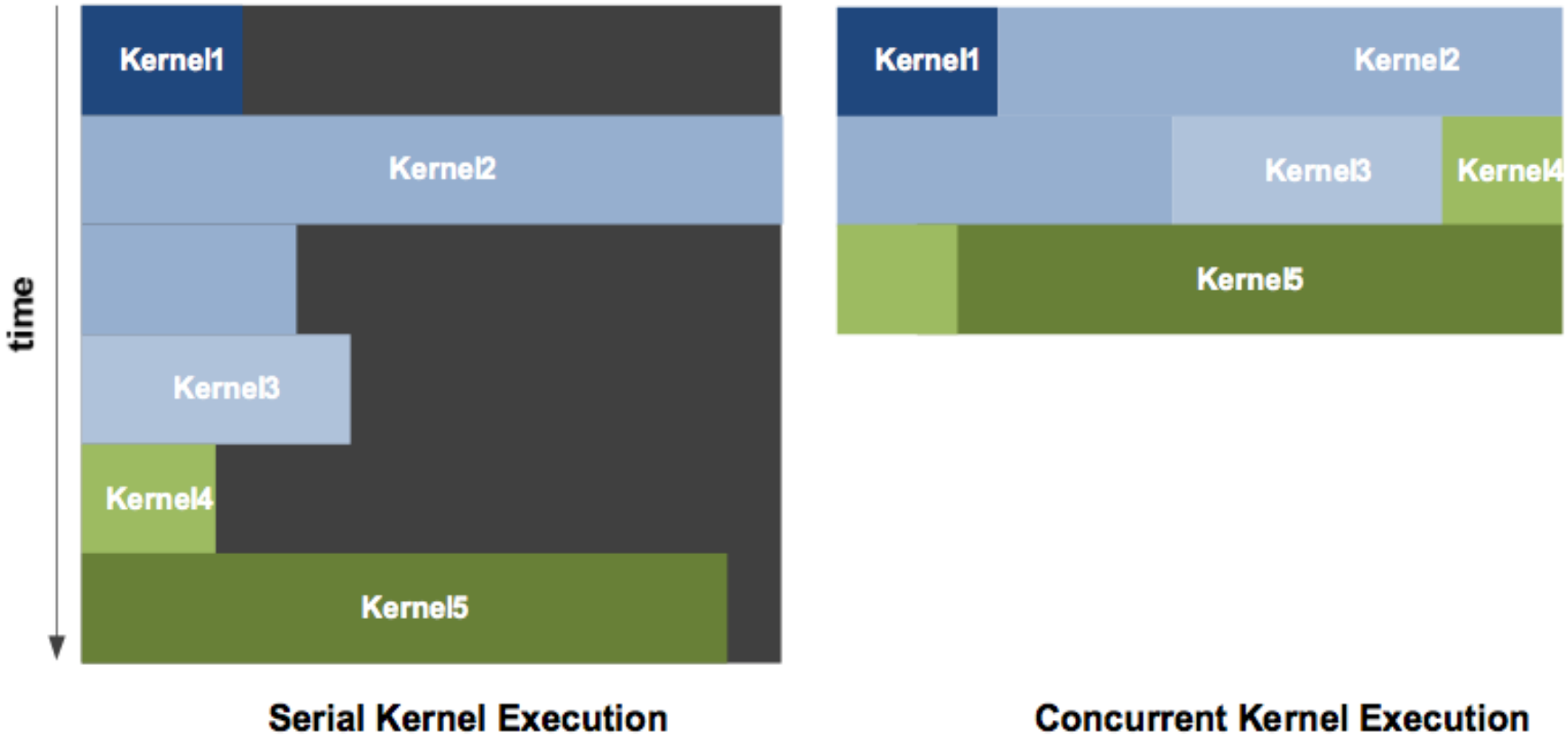
Melhorias em bibliotecas matemáticas

Arquitetura FERMI

Melhoria no desempenho de dupla precisão

Suporte a memória ECC

Suporte a múltiplos kernels simultâneos



Modo de Emulação

Neste tutorial, executaremos os códigos em modo emulação.

Existem 2 motivos principais para utilizar o modo emulação:

- Quando não temos uma placa gráfica compatível :-)
- Para realizar debugging (GDB e printf)

Mas existem grandes desvantagens:

- Desempenho muito inferior
- Difícil avaliar qualidade da implementação
- Pode mascarar bugs no código (especialmente memória)

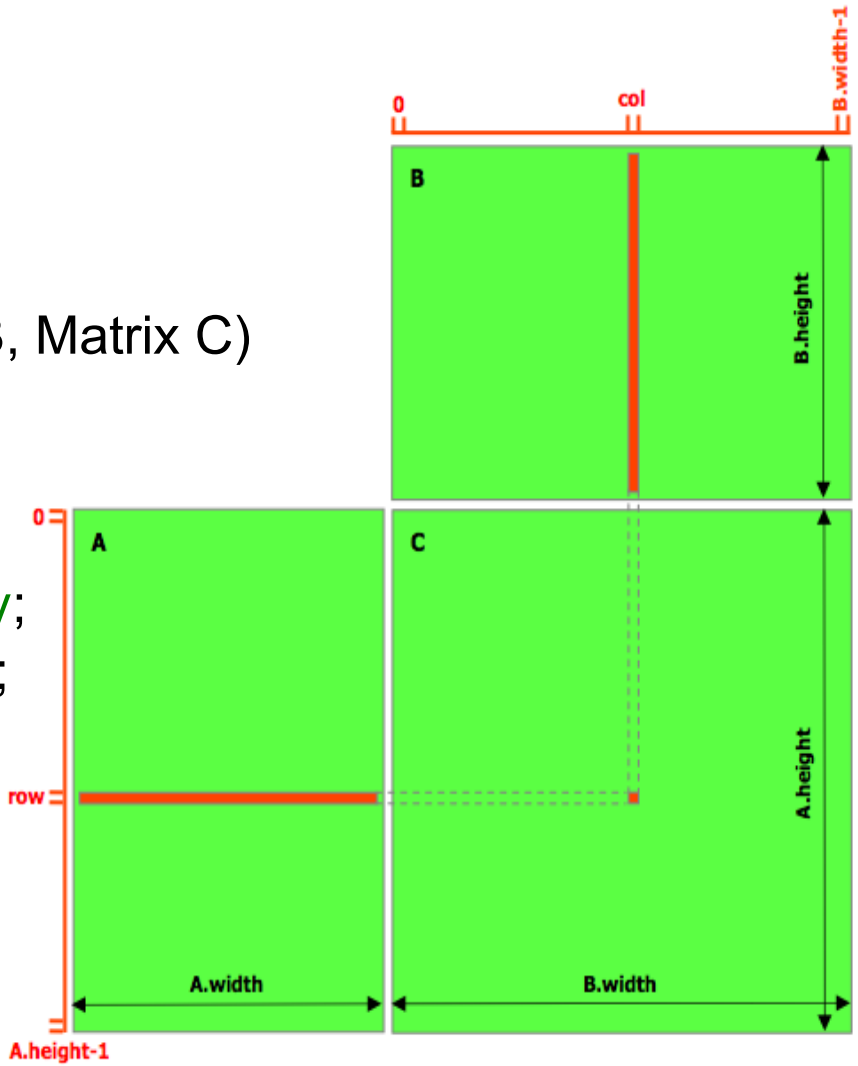
Hoje é possível realizar debugging diretamente na GPU, de modo que a emulação está sendo descontinuada.

Exemplo 2: Multiplicação de Matrizes

```
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width, height;
    float *elements;
} Matrix;

__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;


    C.elements[                 ] = Cvalue;
}
```



Exercício

1) O kernel está correto? Inspecionando o código, procure por erros que impeçam seu funcionamento e o corrija

2) Escreva o código da CPU que chama o kernel.

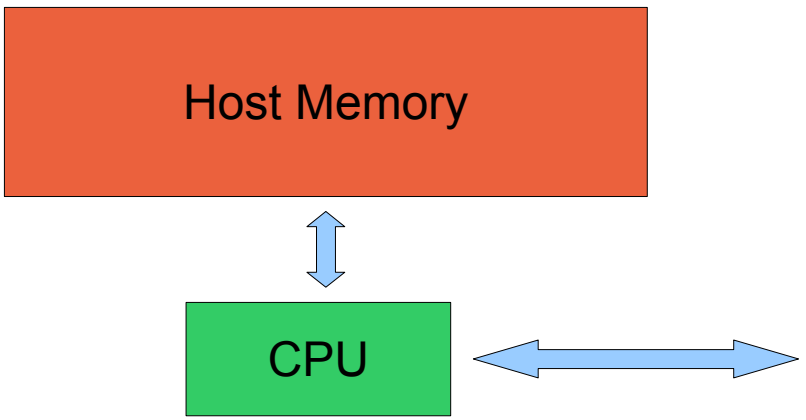
Dica: Use os seguintes comandos para lançar o kernel

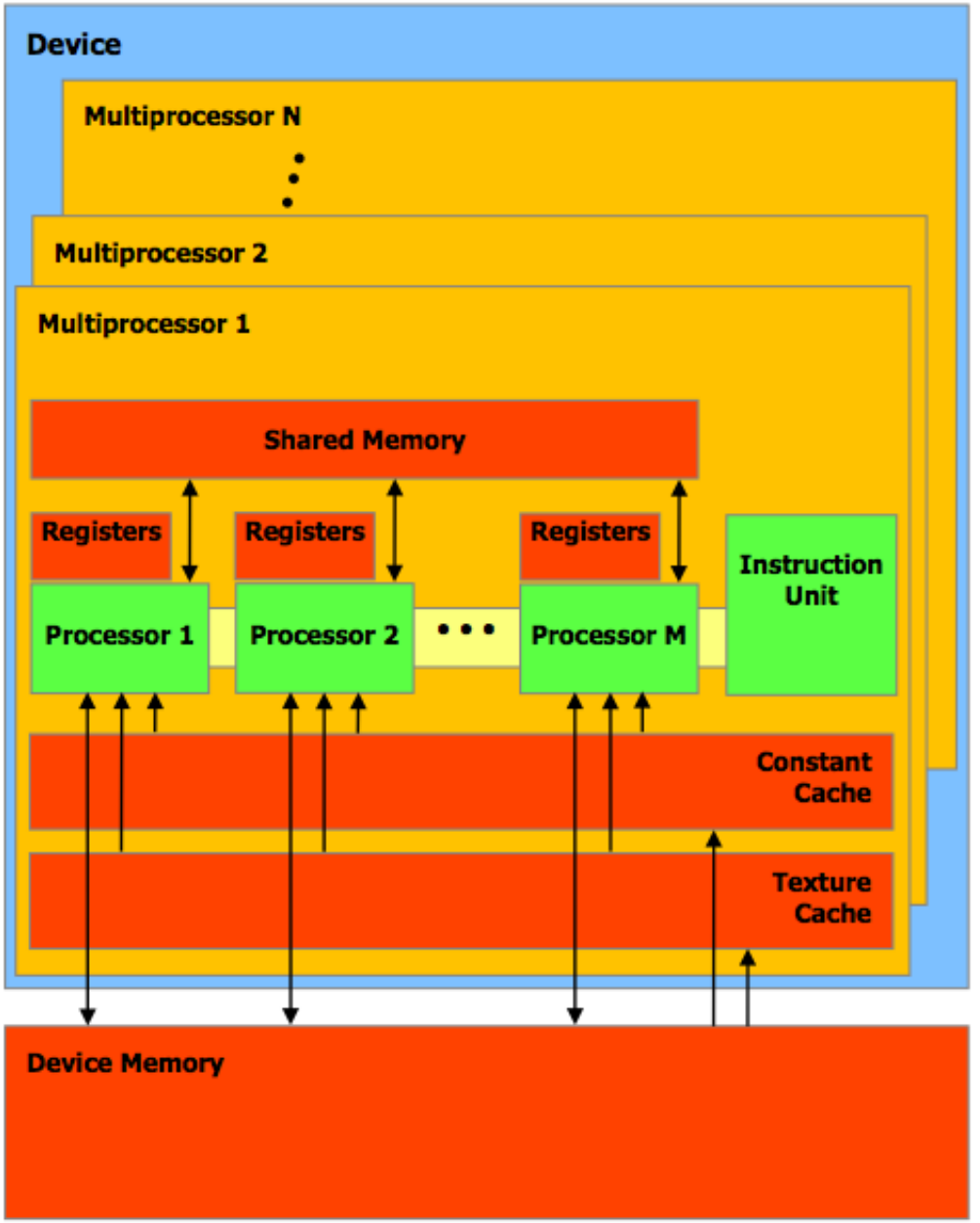
```
dim3 dimBlock(BLOCK_SIZE_X, BLOCK_SIZE_Y);
dim3 dimGrid(GRID_SIZE_X, GRID_SIZE_Y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

3) Escreva um teste para seu exemplo

Uso da Memória Compartilhada

A otimização que traz o maior desempenho é o uso da memória compartilhada de cada multiprocessador
- O acesso à memória global demora centenas de ciclos

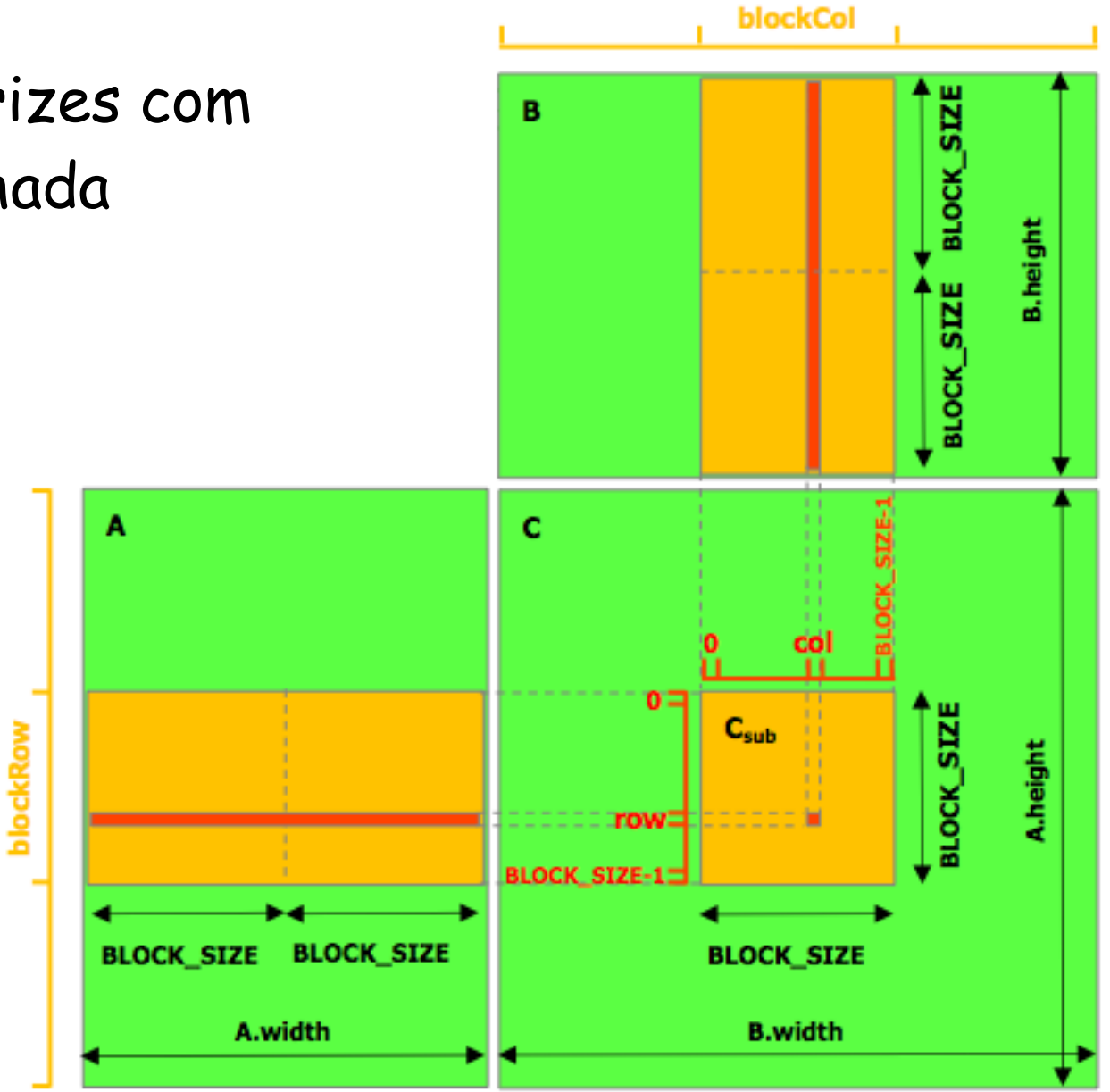




Otimizações de código

Multiplicação de matrizes com memória compartilhada

Pedaços da matriz são colocados na memória de cada multiprocessador



```
// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
```

```
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

Otimizando o código

Otimizar o código é a parte mais difícil e trabalhosa durante o desenvolvimento de um programa CUDA.

Hoje este processo ainda é "artesanal", com a necessidade de tentarmos diferentes estratégias.

Alguns pontos importantes a considerar são:

- Divergência do controle de fluxo
- Ocupação dos processadores
- Acesso combinado (coalesced) à memória global
- Conflitos de bancos da memória compartilhada
- Sobrecarga da chamada do Kernel

Divergência do Controle de Fluxo

As thread de cada bloco são divididas em warps, contendo 16 ou 32 threads

→ GPUs permitem a execução simultânea de todas as threads do warp, desde que todas executem o mesmo código

Quando threads executam códigos diferentes, dizemos que houve uma divergência na execução do código.

Exemplos: comandos if, else, while, for, etc.

```
__global__ void VecAdd(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x;
    if (i < n)
        C[i] = A[i] + B[i];
}
```

Ocupação dos Multiprocessadores

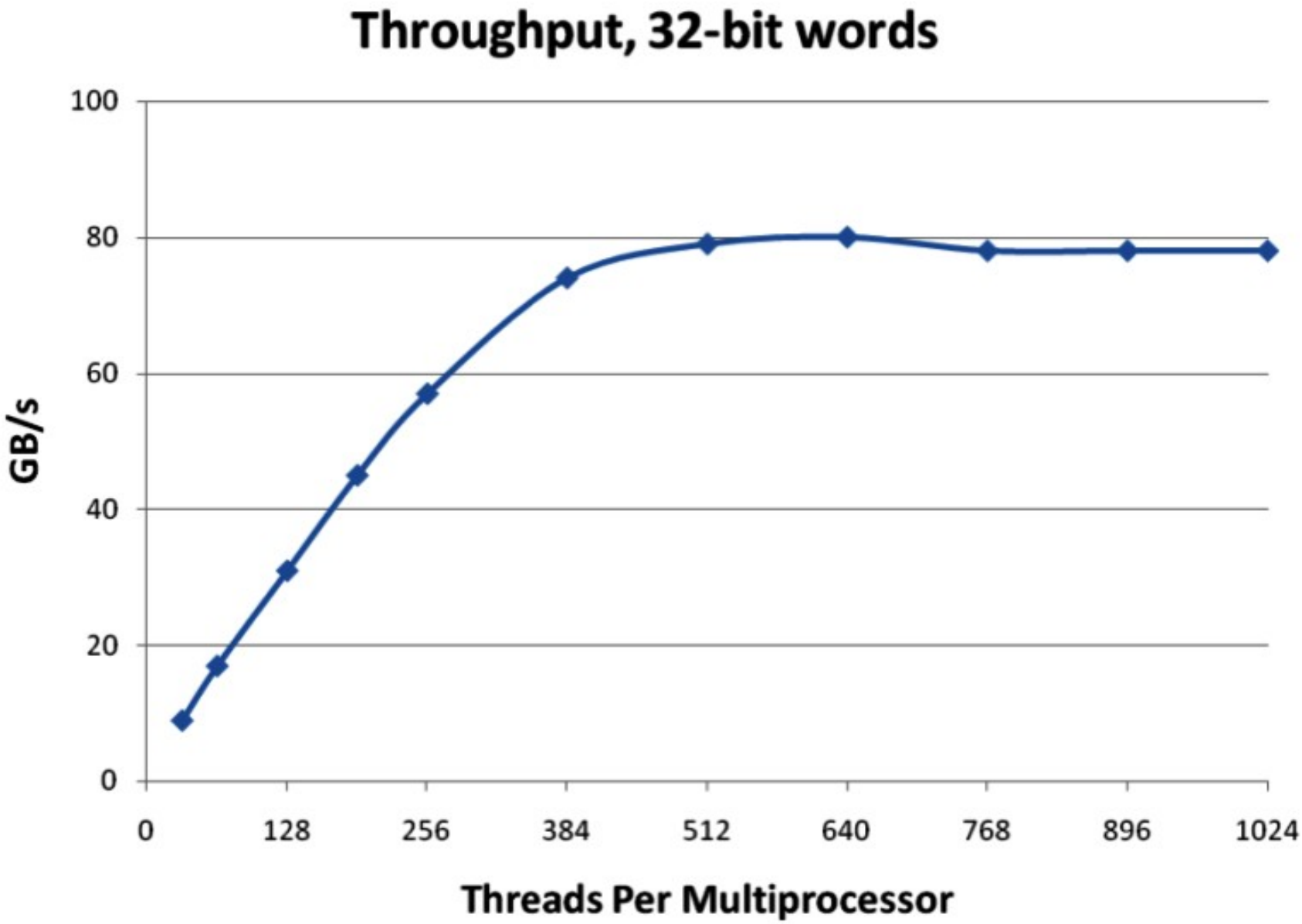
O segredo para obter um bom desempenho é manter os processadores da GPU sempre ocupados. Para tal:

- Os blocos devem ter tamanhos múltiplos de 32
  Assim, coincidem com o tamanho dos warps

- Usar o menor número possível de registradores por thread
  O número de blocos por multiprocessador será maior

Com mais blocos por multiprocessador, haverão mais opções de threads para execução
Especialmente quando as threads estiverem esperando por dados da memória global

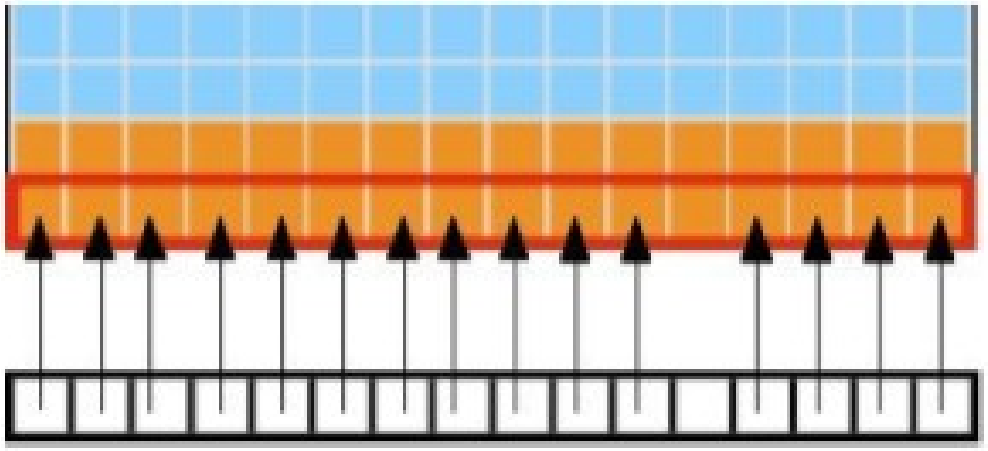
Ocupação dos Multiprocessadores

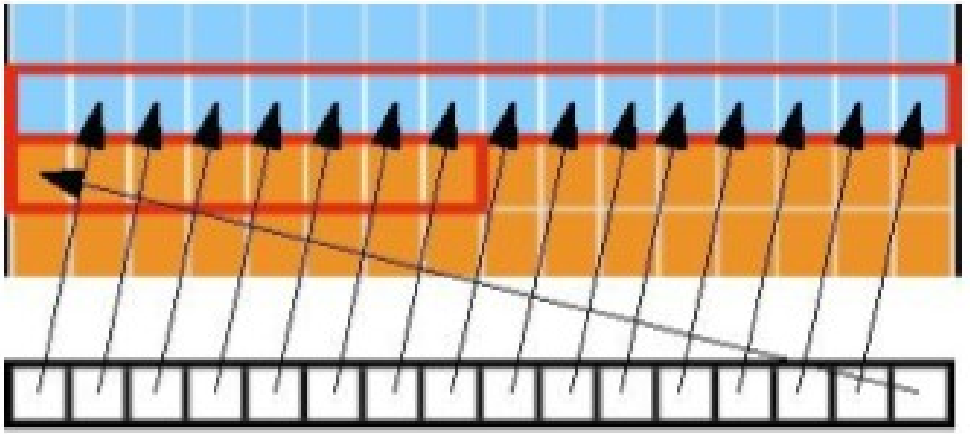


Acesso Combinado (Coalesced)

Quando 16 threads de um warp acessam a memória ao mesmo tempo, CUDA combina os acessos em uma única requisição

Para tal, todas os endereços devem estar localizados em um único intervalo de 128 bytes



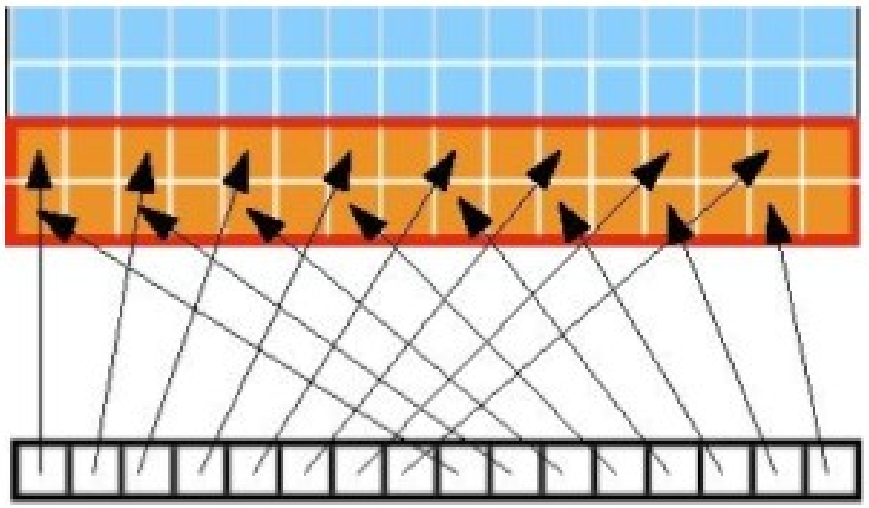


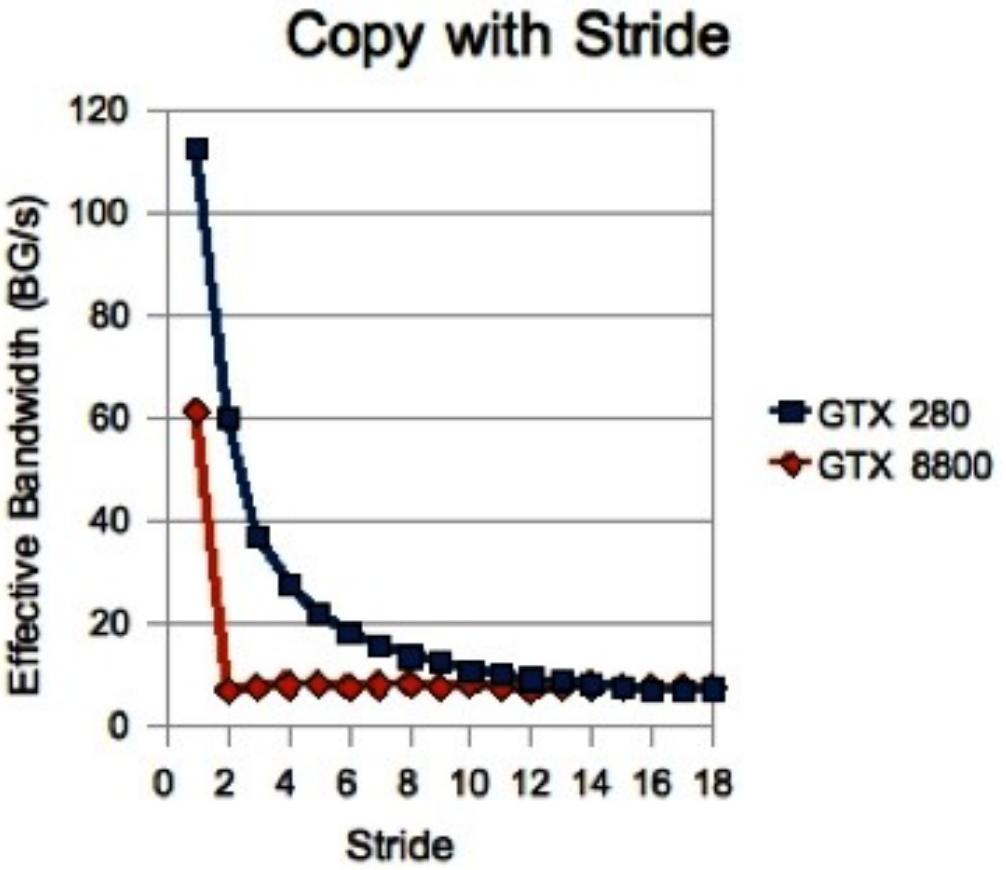
Acesso Combinado (Coalesced)

É melhor que os dados a serem lidos sejam adjacentes

No caso de dados com separações de 2 ou mais posições, o desempenho cai consideravelmente

Dados com separação de 2





Acesso Combinado (Coalesced)

Qual é a melhor opção?

Um array de estruturas?

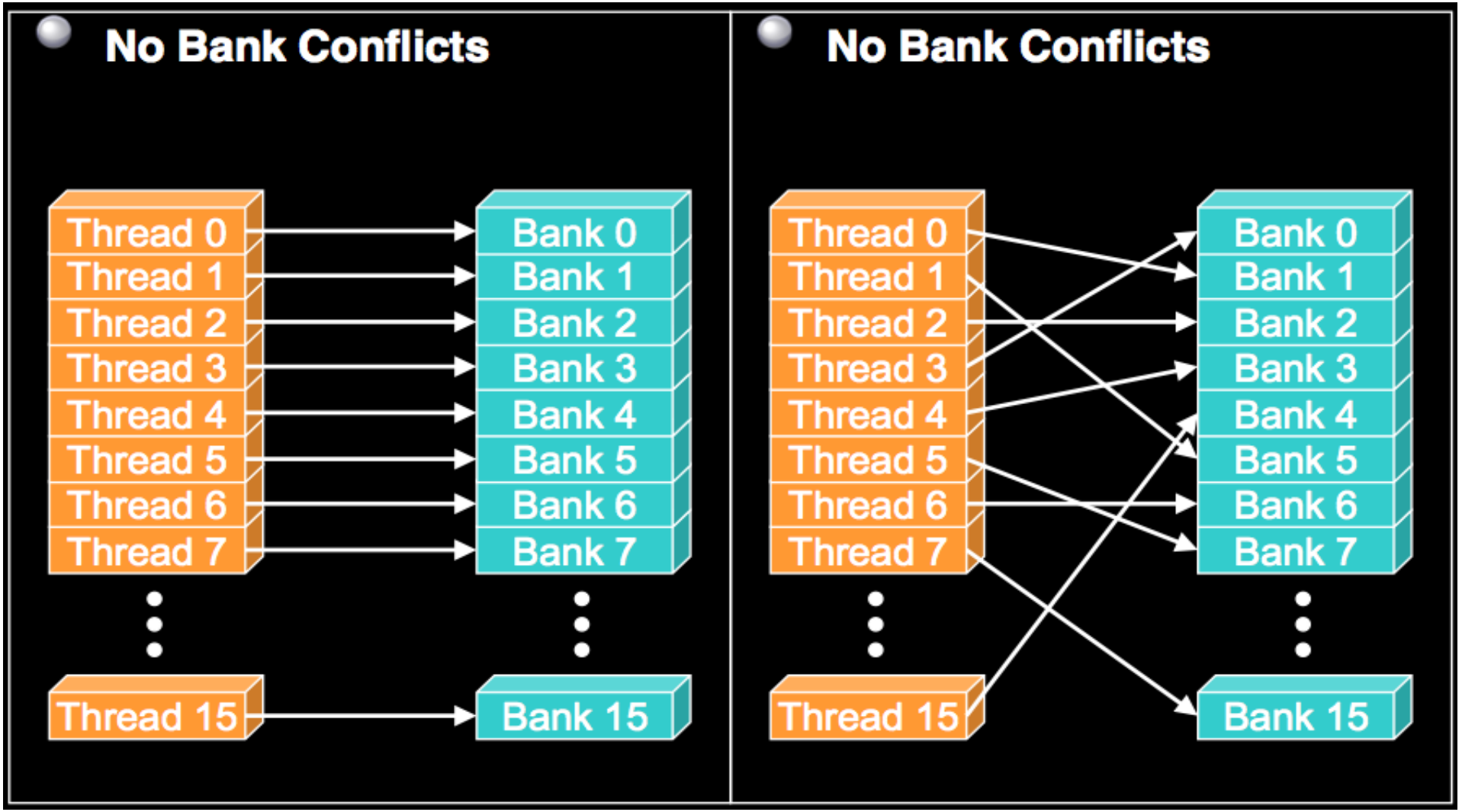
```
struct Dados {
    int chave;
    int valor;
};
Dados *vetorDados;
```

Ou uma estrutura de arrays?

```
struct Dados {
    int *chave;
    int *valor;
};
Dados estruturaDados;
```

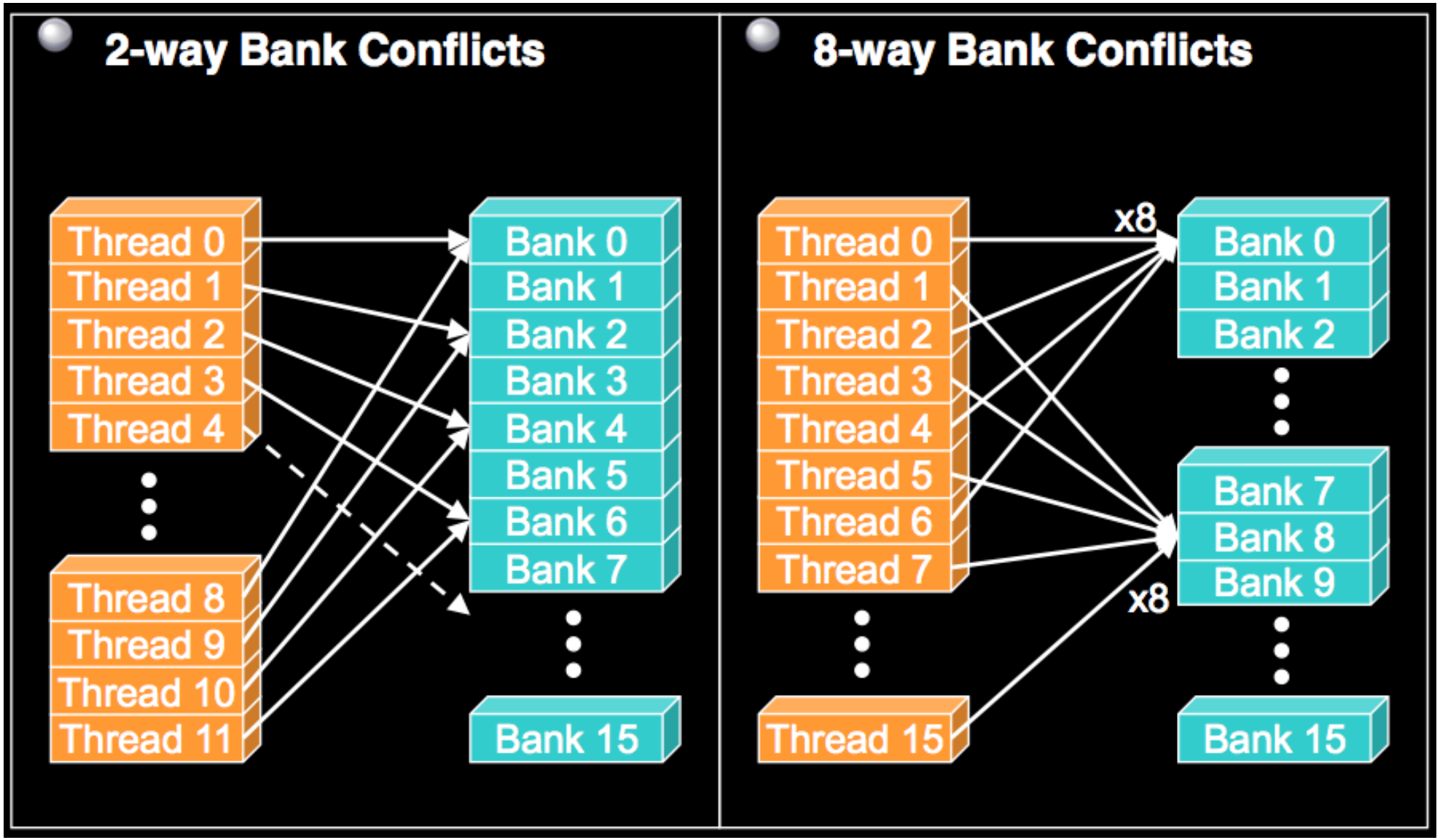
Bancos de Memória Compartilhada

A memória compartilhada é dividida em 16 bancos, com palavras de 32 bits, que podem ser acessados simultaneamente



Bancos de Memória Compartilhada

Quando ocorrem conflitos, a GPU serializa as requisições



Reduza as Chamadas do Kernel

As chamadas ao Kernel causam um overhead

- Além disso, muitas vezes precisamos passar dados para o kernel ou obter resultados da execução

Uma maneira de melhorar o desempenho é agrupar um maior número de tarefas em uma chamada

Exemplo: Se for preciso resolver diversos sistemas lineares, é melhor resolver todos em uma única chamada do kernel

Parte III

FireStream, OpenCL e Bibliotecas para CUDA

NVIDIA Performance Primitives (NPP)

Biblioteca de funções utilizáveis em aplicações para GPUs

No momento possui primariamente funções para processamento de imagem e vídeo. Exemplos:

- Filtros
- Redimensionamento
- Histogramas
- Aplicação de limiares
- Detecção de bordas, etc.

GPU Accelerated Linear Algebra

Biblioteca de funções para álgebra linear. Exemplos:

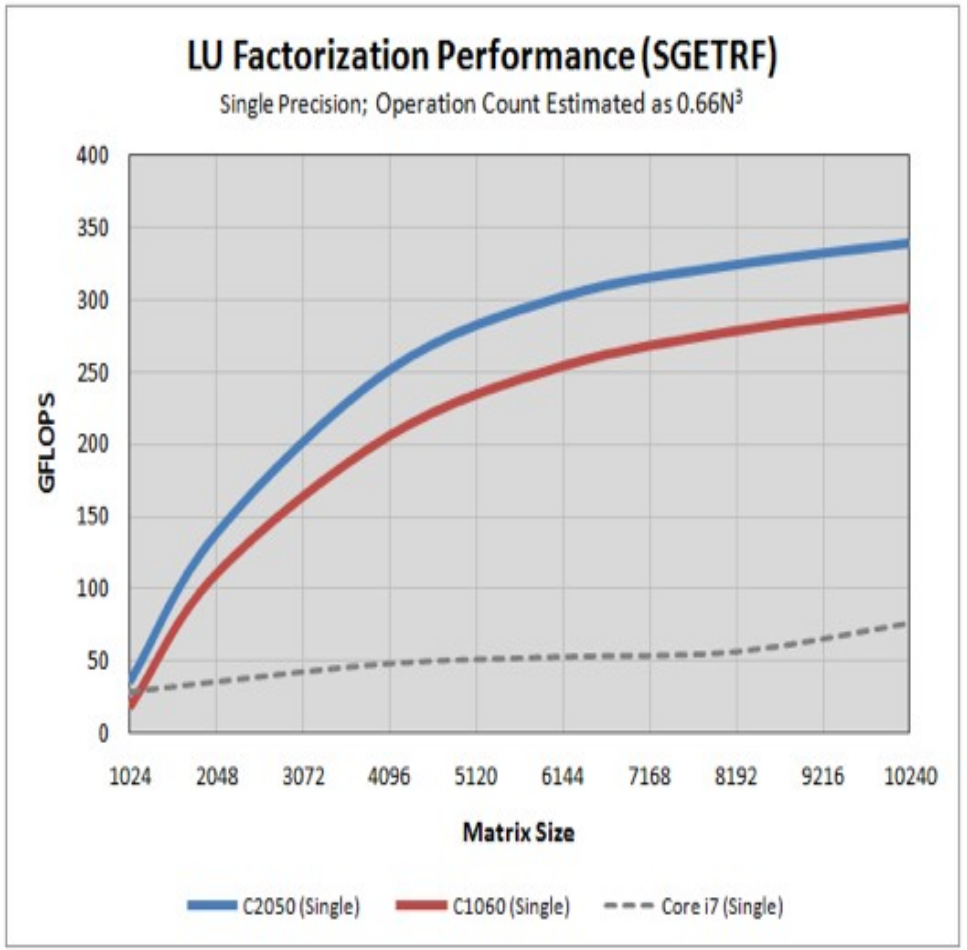
- Solução de sistemas lineares
- Rotinas para autovalores
- Triangularizações
- Decomposição em valores singulares, etc.

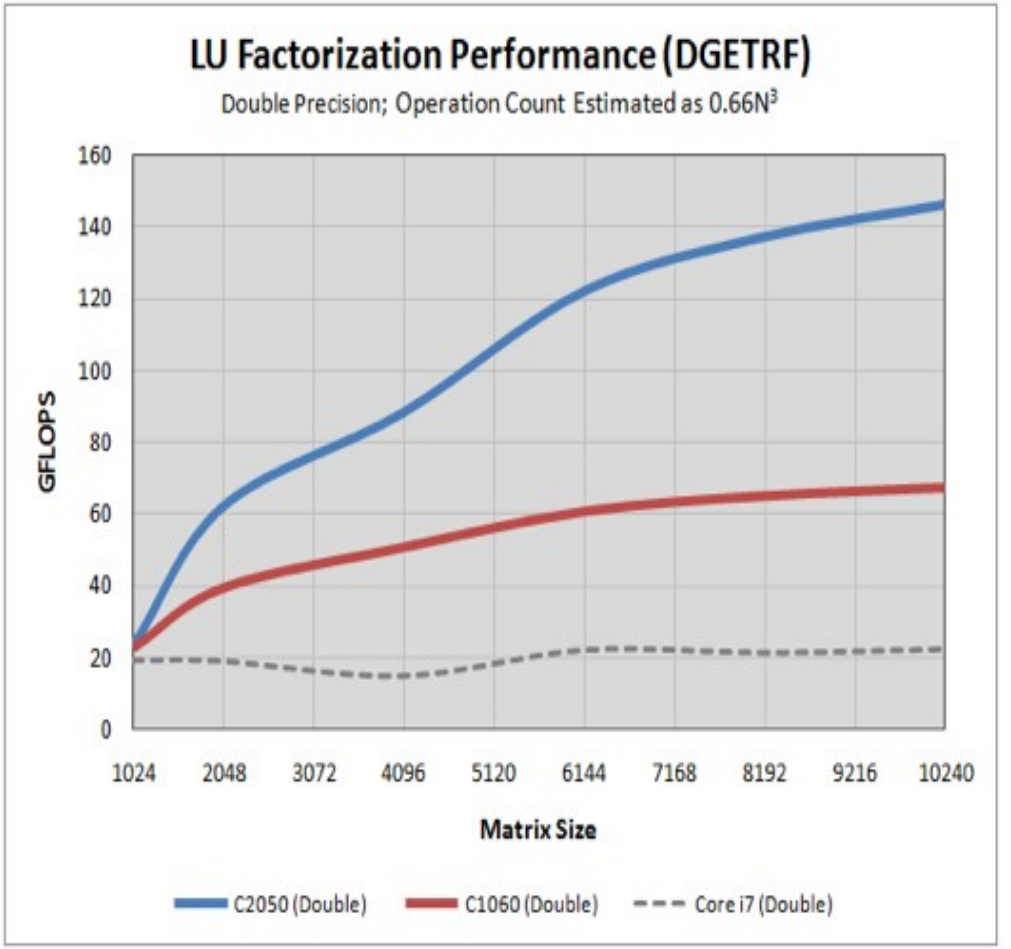
Interfaces com C/C++, Fortran, Matlab, Python:

- Chamadas realizadas diretamente da aplicação para CPU
- Não é preciso programar em CUDA

Desempenho

Aplicação de Fatorização com precisão simples e dupla





AMD FireStream

As placas da AMD (antiga ATI) também fornecem suporte a GPGPU, através das placas da Série FireStream

FireStream 9270

- 1 GPU com 800 núcleos
- 2GB RAM, 108 GB/s
- 1.2 TFlop (single), 240 GFlop (double)

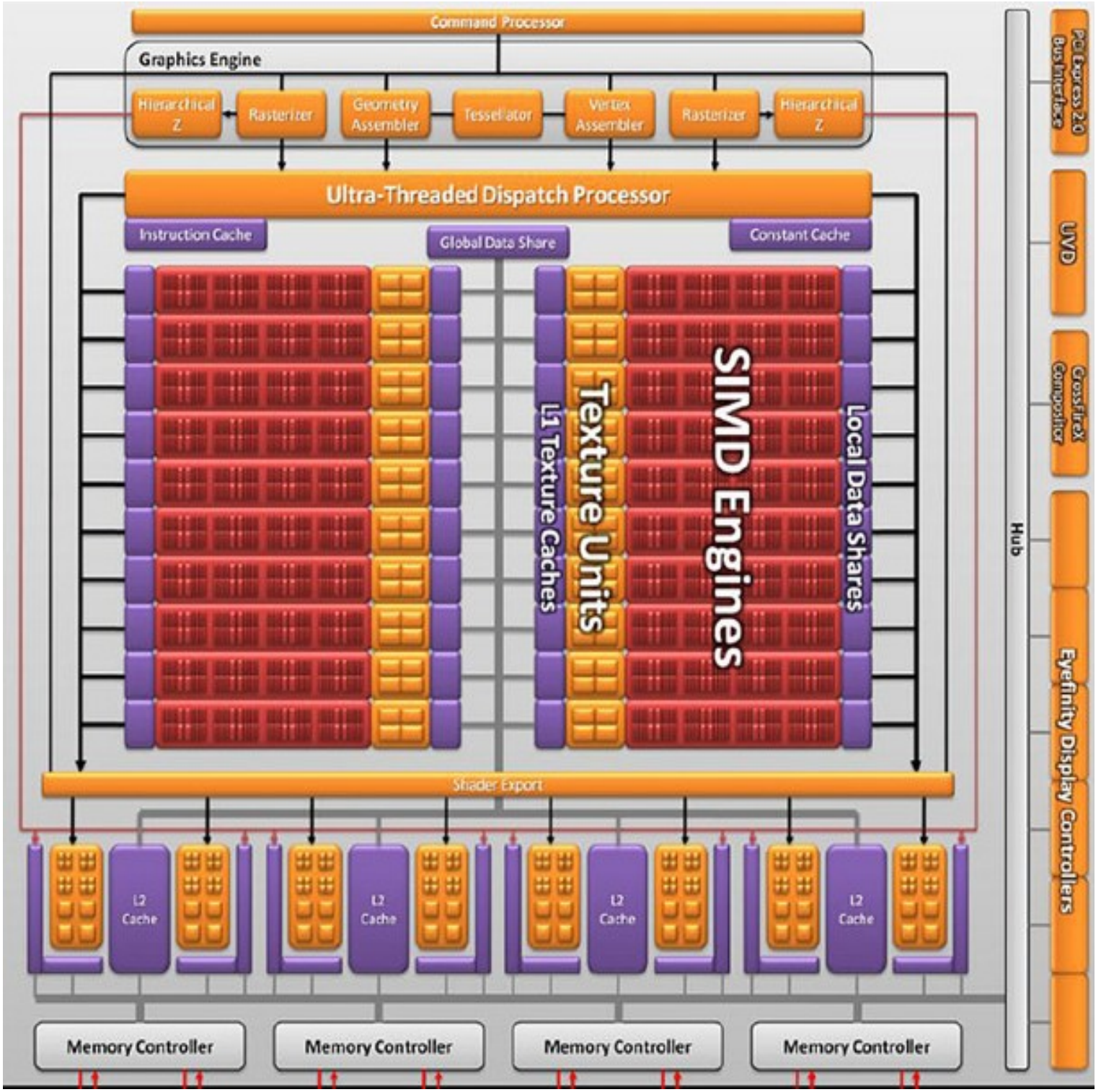
3o trimestre/2010

- FireStream 9370 e 9350
- 4GB RAM
- 2.64 TFlop (single), 528 GFlop (double)

Cyprus Architecture

10 SIMD units with 16 stream cores with 5 ALUs on each = 1600 ALUs

- Difícil de usar as 5 ALUs
- Relógio das ALUs é igual ao do chip

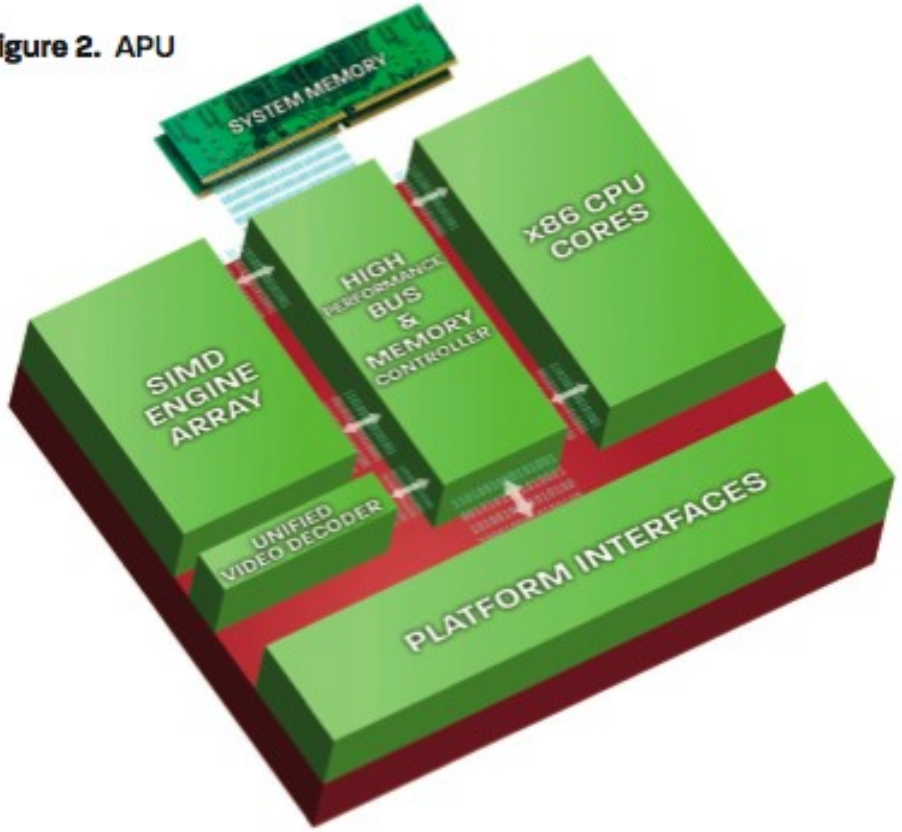


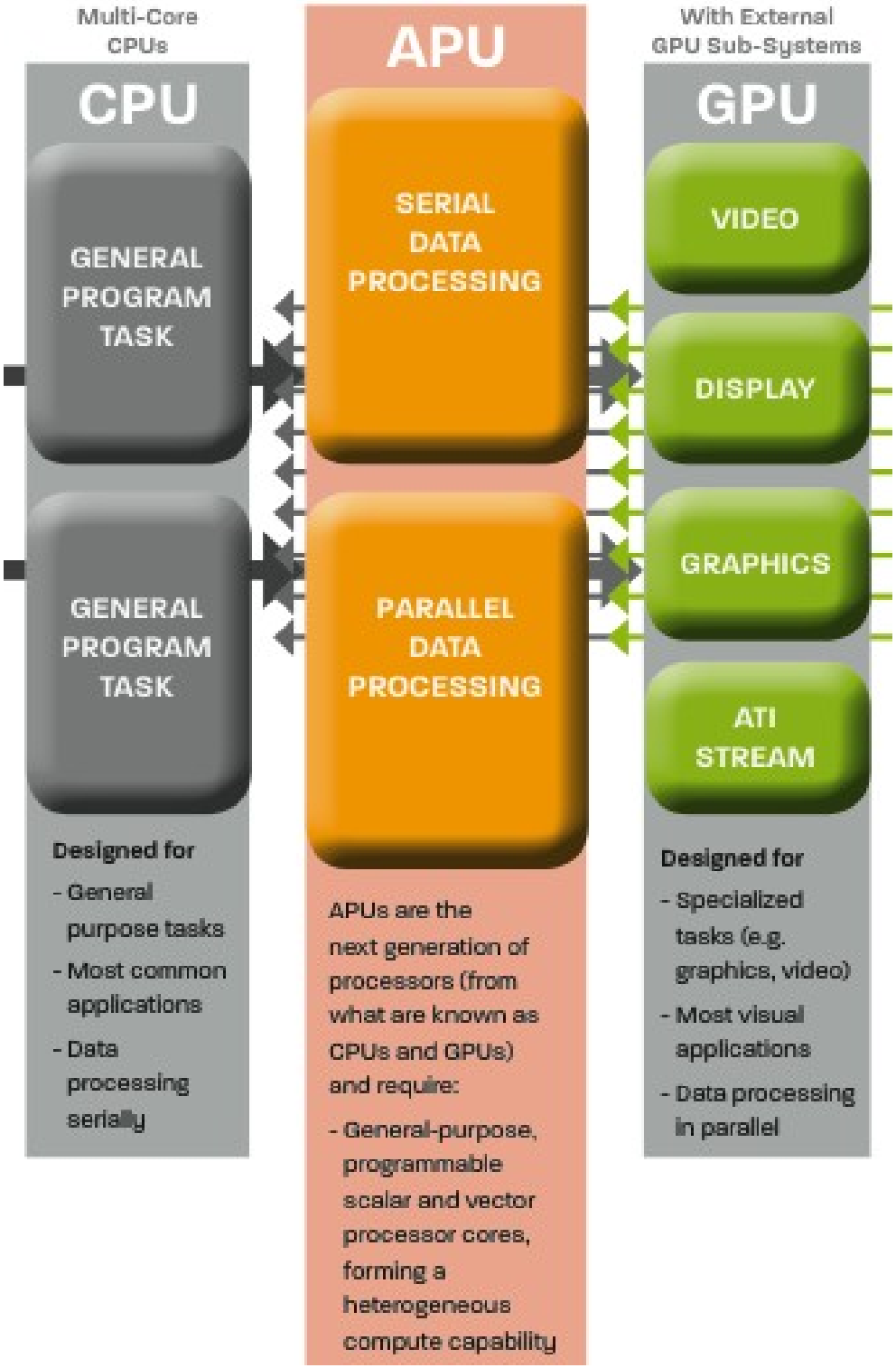
AMD Fusion

Irá combinar em um chip:

- Núcleos x86
- Núcleos SIMD para processamento paralelo

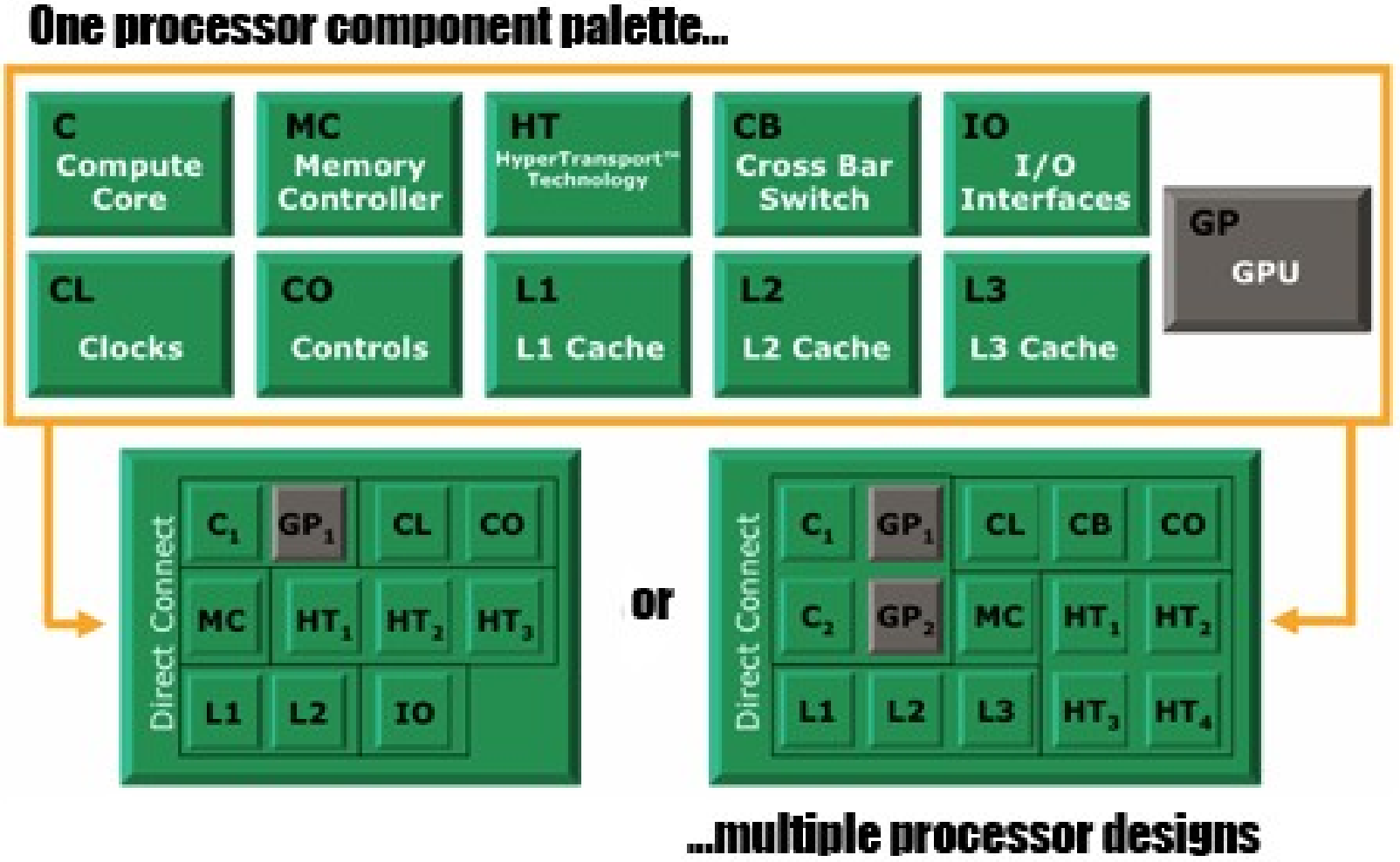
Figure 2. APU





APU (Accelerated Processing Unit)

Diferentes tipos de configurações poderão ser criadas a partir de blocos básicos.



ATI Stream SDK

O acesso ao poder computacional é feito através do uso da ATI Stream SDK.

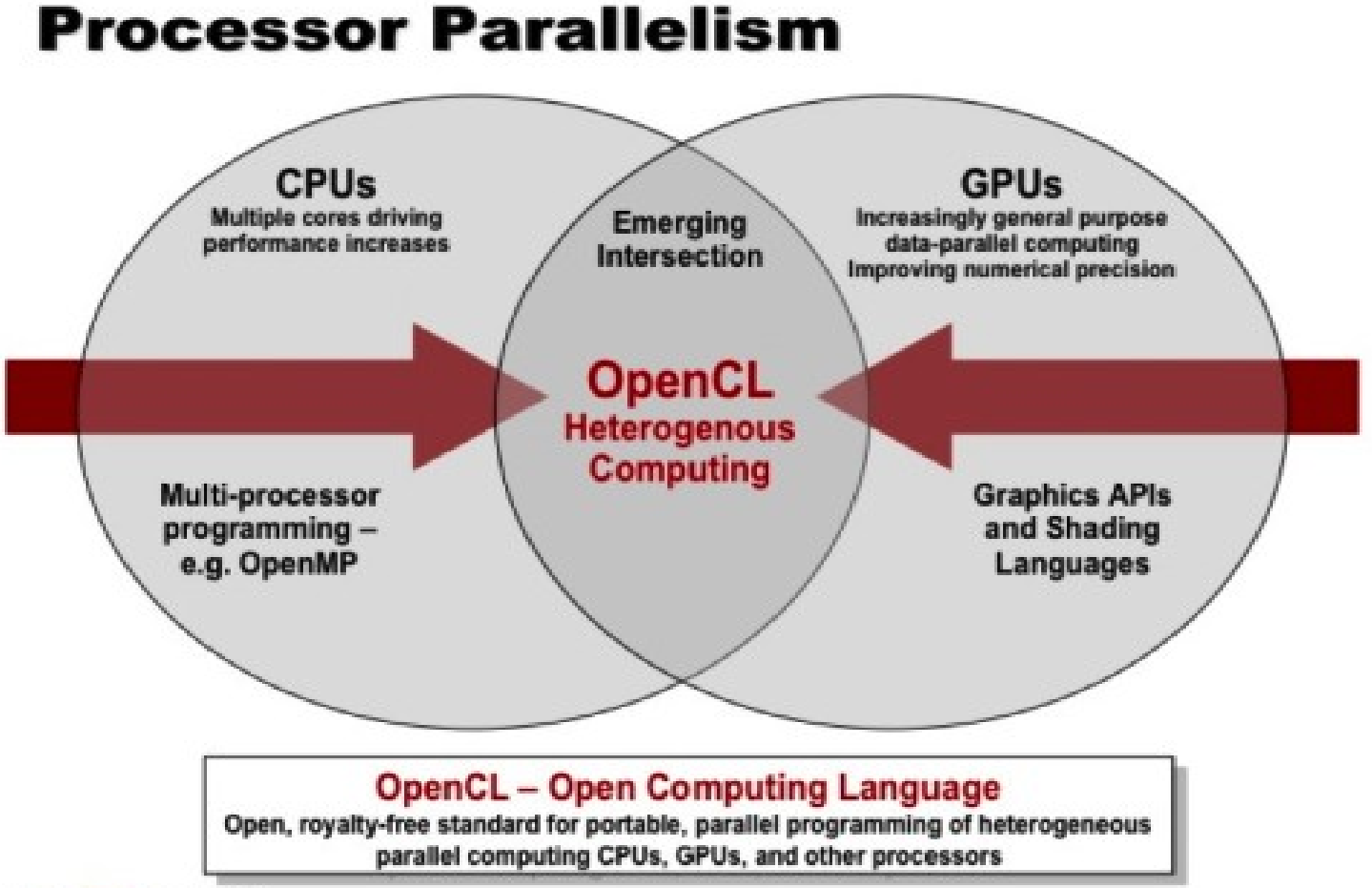
Compatível com as arquiteturas FireStream e Fusion

Programação através da linguagem OpenCL 1.0
- Anteriormente utilizava o Brooks+

OpenCL

Diferentes hardwares utilizam diferentes plataformas:

- CUDA para nVIDIA e Brooks+ para AMD/ATI
- OpenMP para programação de múltiplos processadores



OpenCL

O objetivo é permitir que um mesmo programa possa ser executado em múltiplas plataformas de hardware.

- Padrão aberto, totalmente livre de royalties.
- Suporte da Apple, nVIDIA, AMD, Intel e outras.
- Falta a Microsoft, que tem sua solução própria :-)
  - DirectCompute, que faz parte do DirectX 10 e 11
  - Alguém lembra da disputa DirectX vs OpenGL?

OpenCL

Apesar das vantagens, existe um certo ceticismos sobre a adoção do OpenCL

- Diferentes arquiteturas tem características específicas, que precisam ser levadas em conta na otimização

- Programas genéricos acabariam sendo nivelados por baixo

- Quando comparado com CUDA, a sintaxe de OpenCL é mais complicada, dado que ela precisa trabalhar com múltiplas plataformas

Parte IV

Aplicação:
Simulação de Redes Neuronais

Neurociência Computacional

Hoje temos a visão macro e micro do cérebro, mas ainda não somos capazes de ligar adequadamente estas visões

Objetivo da neurociência computacional:

Entender como um conjunto de neurônios e sua conectividade geram funções cognitivas complexas.

Para tal, criamos modelos matemáticos de áreas cerebrais e simulamos estes modelos em computadores

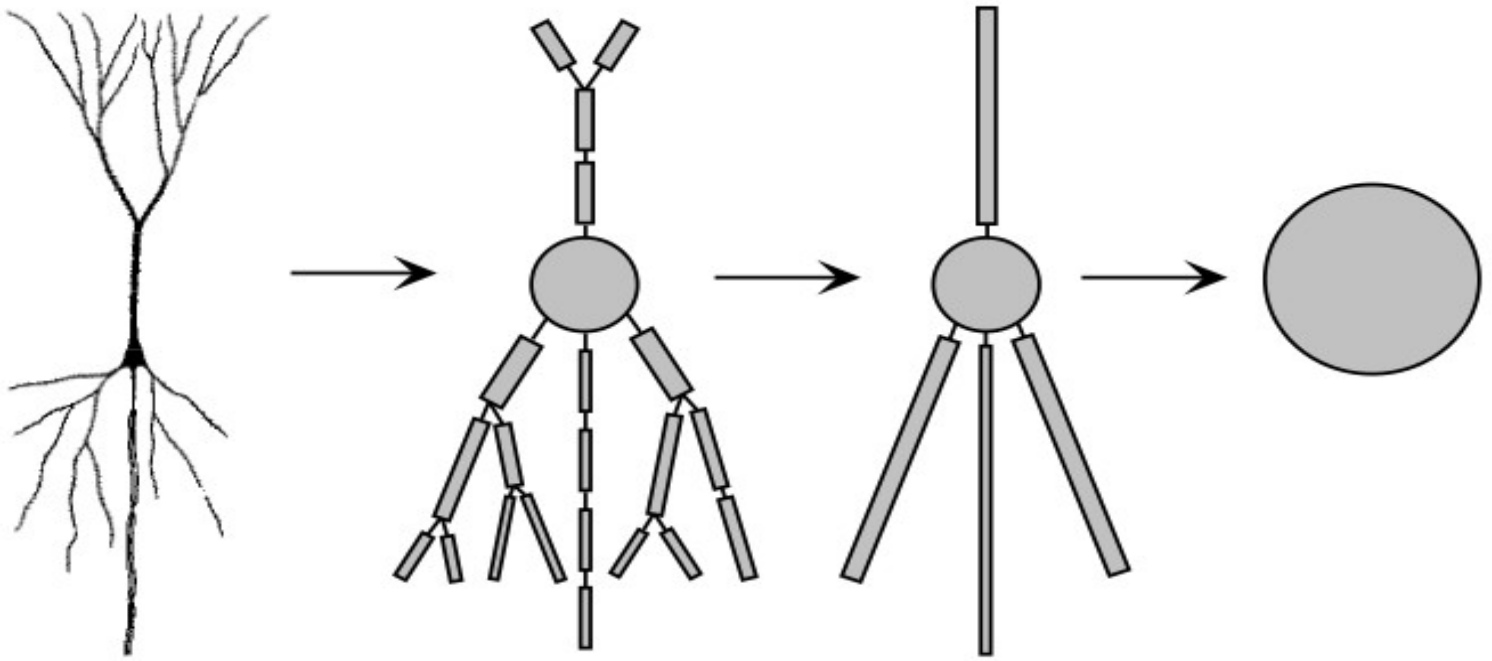
Modelos podem ser compostos por até milhões de neurônios

Simulação exige um grande poder computacional provindos de grandes aglomerados ou supercomputadores

Modelagem de Neurônios

Neurônios são um tipo altamente especializado de célula

- Possuem corpo celular, dendritos e axônio
- Mantém um potencial de membrana
- Se comunicam com outros neurônios



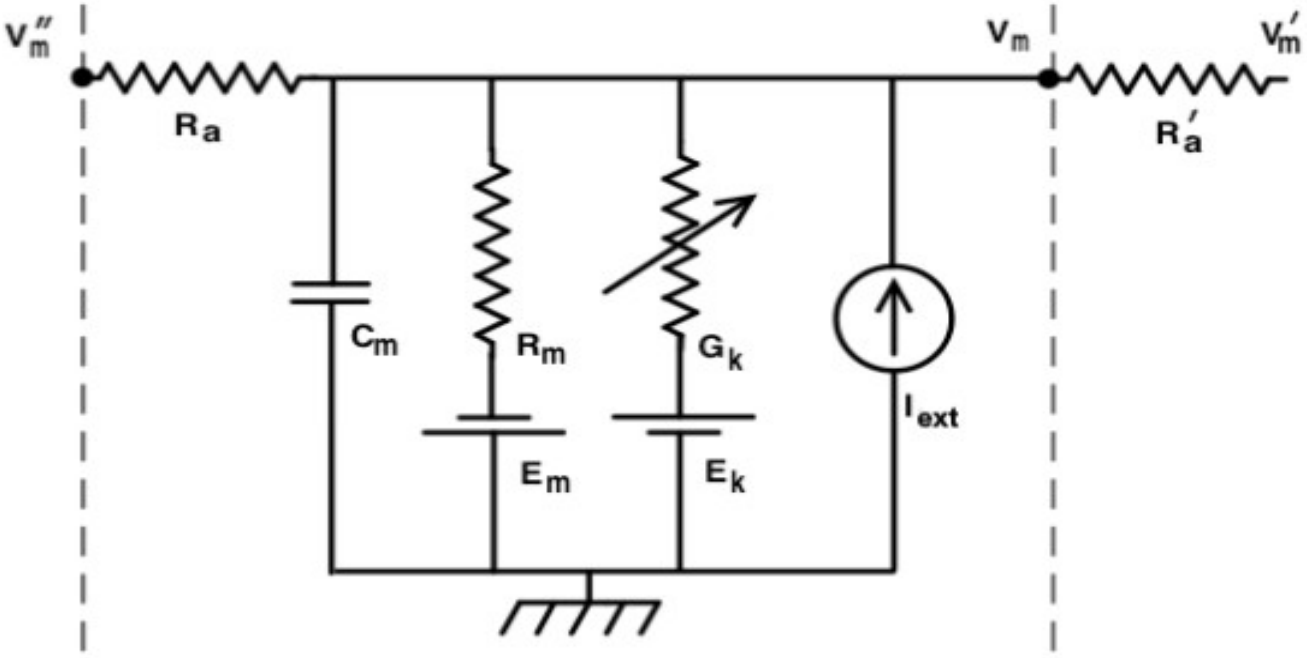
Fonte: Dayan and Abbott. Theoretical Neuroscience. MITPress

Modelagem de Neurônios

Cada compartimento é modelado por um circuito elétrico

Capacitor
- membrana

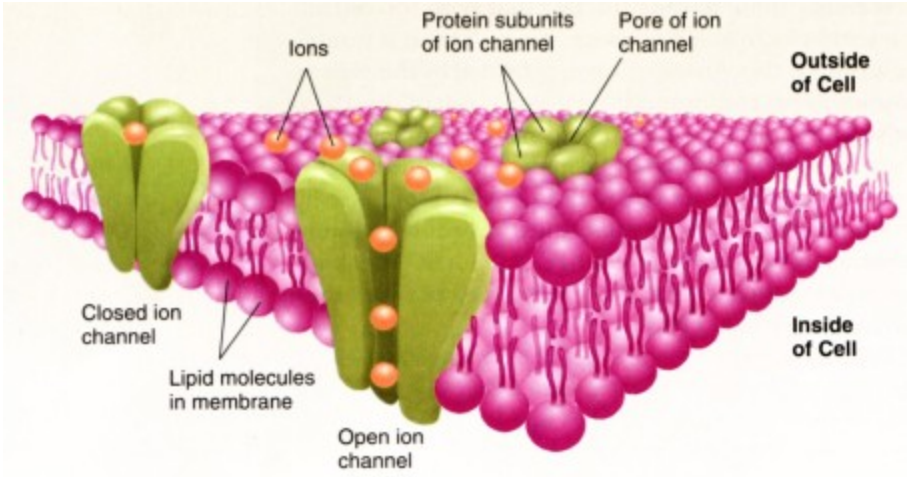
Resistências
- membrana
- canais iônicos
- ligações entre compartimentos

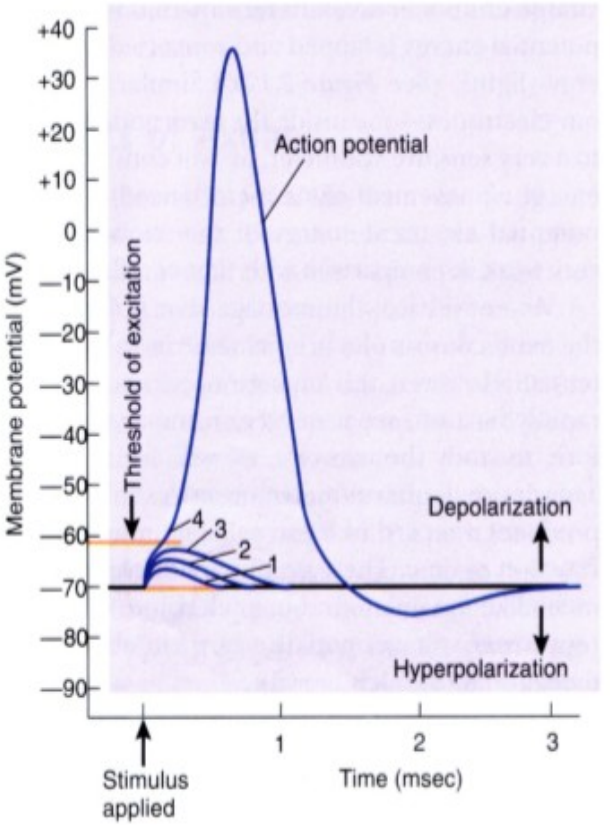


$$\frac{V_{j+1} - 2V_j + V_{j-1}}{R_a} = C_m \frac{dV_j}{dt} + \frac{V_j}{R_m} + I_j.$$

Canais Iônicos Ativos

Canais iônicos ativos: Canais dependentes da voltagem que permitem a geração de potencias de ação





Fonte: N. Carlson, Physiology of Behavior Pearson Ed.

São constituídos por portões que possuem uma dinâmica de abertura e fechamento dependente da voltagem:

$$I_{ion} = \bar{g}_{Na}\, m^3\, h\, (V_m - E_{Na}) + \bar{g}_K\, n^4\, (V_m - E_K) + \bar{g}_L\, (V_m - E_L),$$

$$\frac{dm}{dt} = \alpha_m(V)\,(1-m) - \beta_m(V)\, m,$$

Integração Numérica

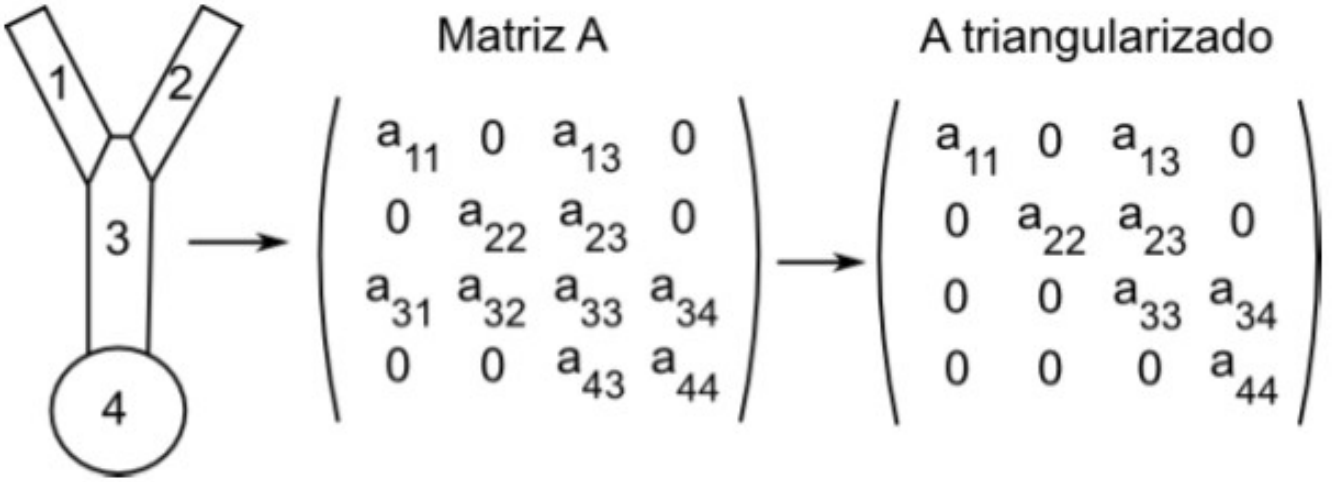
É preciso integrar uma equação diferencial por neurônio

Integração implícita gera melhor estabilidade

$$\frac{V_{j+1} - 2V_j + V_{j-1}}{R_a} = C_m \frac{dV_j}{dt} + \frac{V_j}{R_m} + I_j.$$

$$[\ A\ ] \times [\ V(t+dt)\ ] = [\ C\ ]$$

Através de uma cuidadosa numeração dos compartimentos e uma triangularização, a solução do sistema pode ser eficiente



Simulação utilizando GPUs

Programas que conseguem bons *speedups* em GPUs:

Subdivido em pequenos subproblemas, que são alocados a diferentes blocos e threads

Cada thread mantém uma pequena quantidade de estado

Alta razão (operações de ponto flutuante) / (memória)

Os subproblemas são fracamente acoplados

Característica de nosso problema

Dezenas de variáveis de estado por thread

- Cada thread deve representar um neurônio
- Limite no número de neurônios por bloco (entre 64 e 128)

Acoplamento entre neurônios durante a comunicação

- Padrão de conexões pode ser irregular
- Atraso no tempo de comunicação entre neurônios

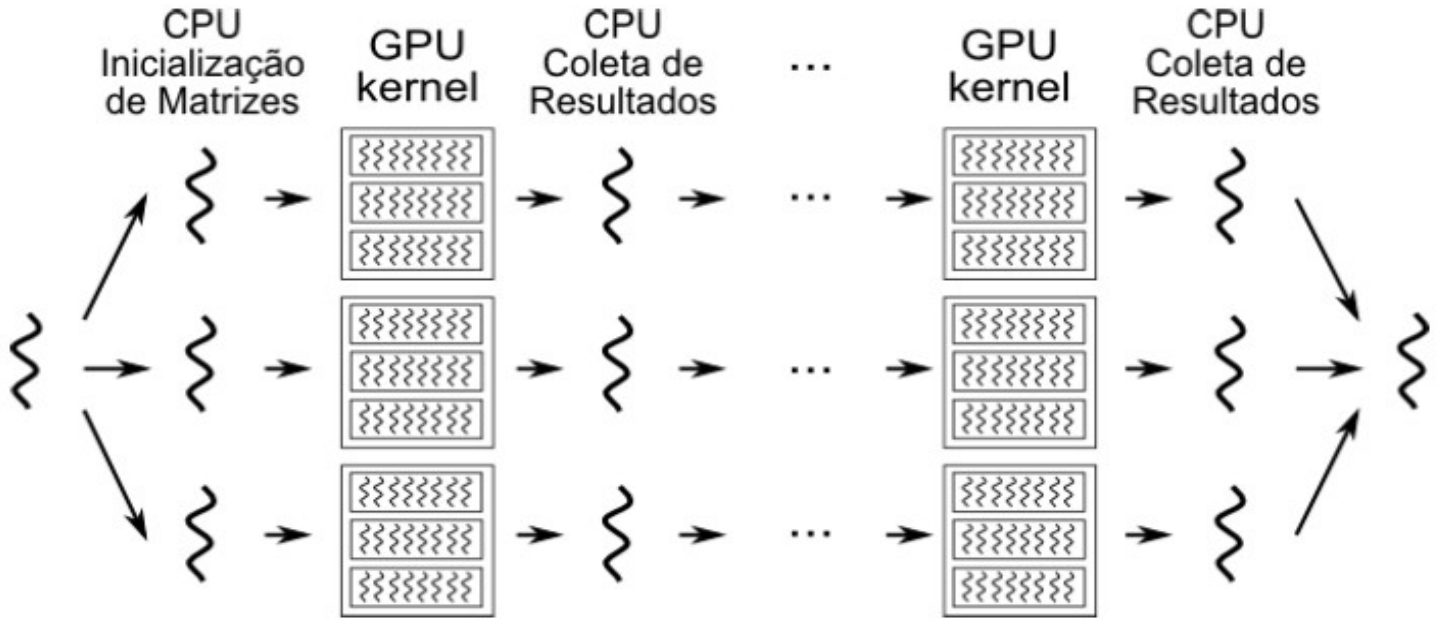
Neurônios são diferentes entre si

- Mas podem ser categorizados em diferentes classes

Implementação do Simulador

A simulação é divididas em rodadas controladas pela CPU:

Inicialização da simulação
Enquanto (tempoAtual < tempoTotal)
    Lança execução do kernel
    Coleta de resultados da simulação
    Envio de dados à GPU
Fim da simulação



Implementação do Kernel

Cada neurônio é implementado por uma thread distinta

- Esta thread executa a diagonalização da matriz e faz substituição dos valores finais

Código para a execução de cada neurônio é igual, de modo que não há divergência de execução

Versão para CPU portada para GPU:

- Usava dados diretamente da memória global.
- O desempenho era ruim, inferior ao da CPU

Otimizações

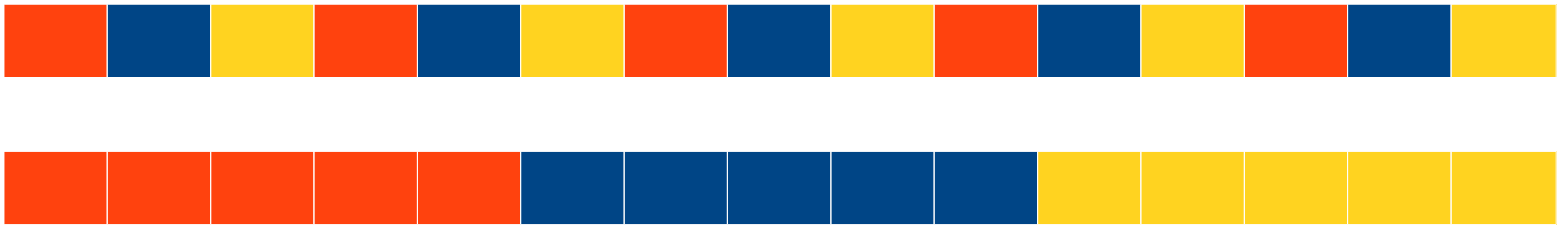
→ Uso da memória compartilhada

Memória compartilhada é pequena para armazenar dados de todos os neurônios de um bloco

Armazenamos os dados que são utilizados múltiplas vezes por passo de integração (leitura e escrita)

→ Leituras conjunta (coalesced)

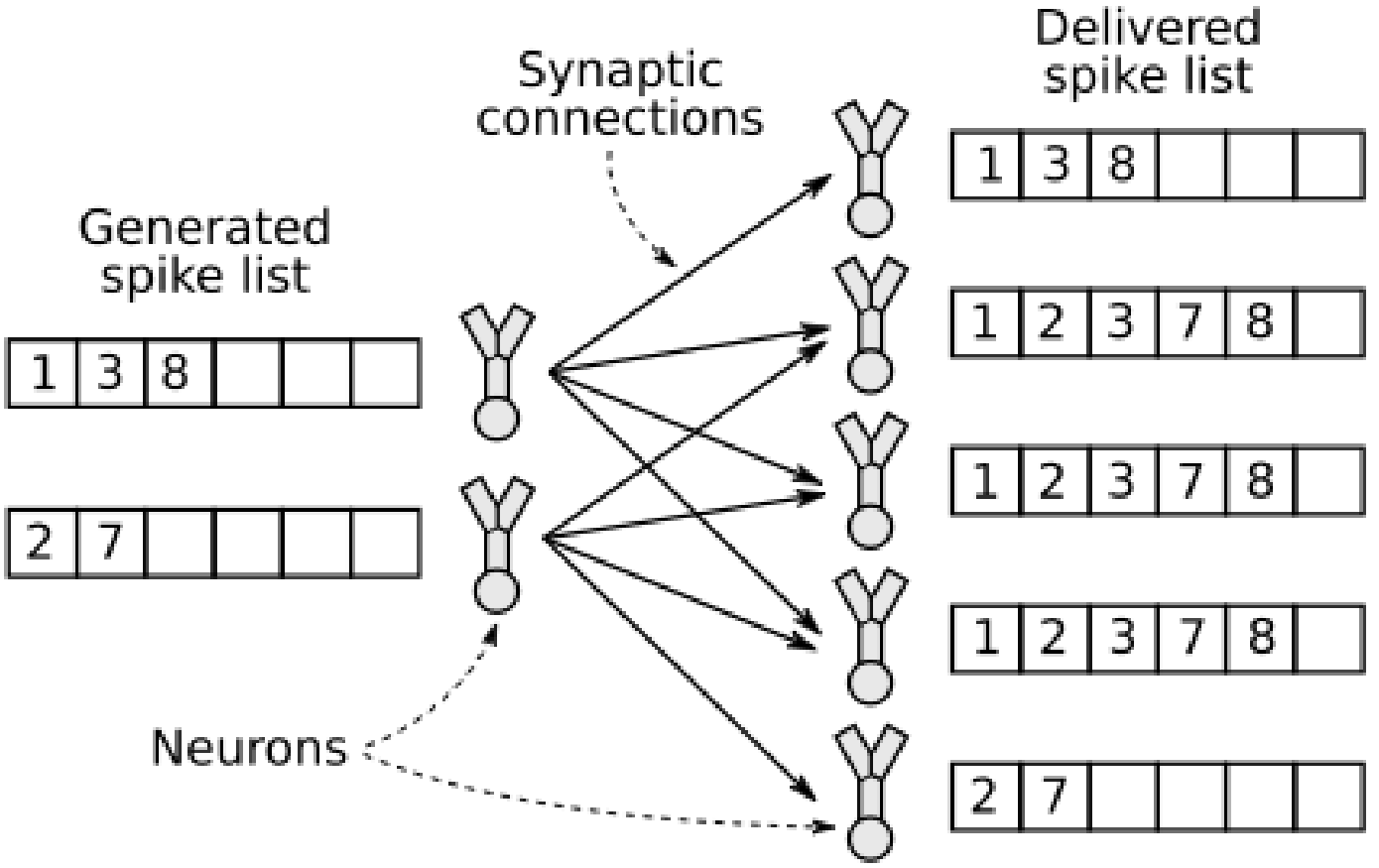
Organização dos dados de modo que leitura de dados de múltiplos neurônios seja feita de modo simultâneo



Comunicação entre Neurônios

Neurônios geram spikes, que são entregues a outros neuônios

Cada neurônio se conecta a centenas ou milhares de neurônios



Comunicação entre Neurônios

Hoje a comunicação é coordenada pela CPU

Gera um grande gargalo no desempenho

Desafio: Comunicação utilizando GPU

1 GPU: Comunicação entre threads de diferentes blocos

Tem que ser feita pela memória global

Uso de primitivas como AtomicAdd()

2 ou mais GPUs em 1 computador:

É preciso passar necessariamente pela CPU

2 ou mais GPUs em múltiplos computadores:

É preciso passar necessariamente pela rede

Hardware Utilizado

Computador Intel Core i7 920, de 2.66GHz,
6 GB de memória RAM
2 duas placas gráficas nVidia GTX 295,
com 2 GPUs e 1892 MB em cada

Ubuntu 9.04 de 64 bits

CUDA versão 2.2, com
drivers 185.18.14.

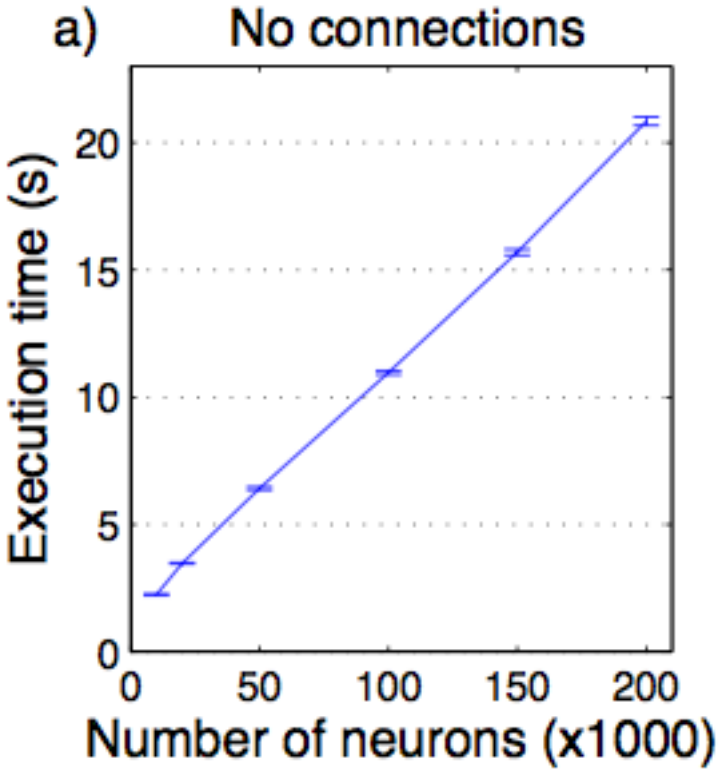
Compilador g++, com
opção -O3.

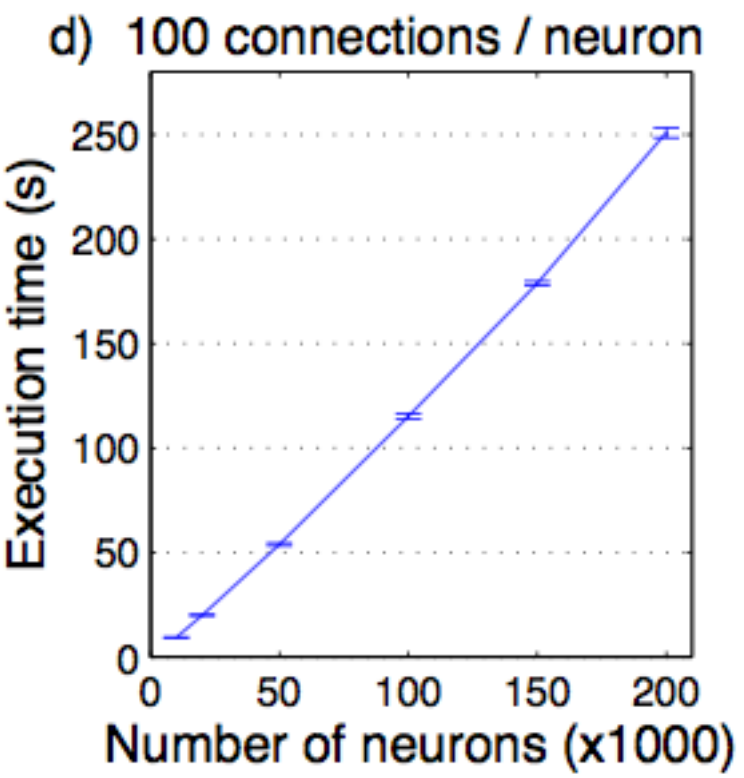
Resultados: Tempo de execução

Cresce linearmente com o número de neurônios

Mas quando incluímos conexões → tempo 10 vezes maior

Comunicação entre neurônios é coordenada pela CPU

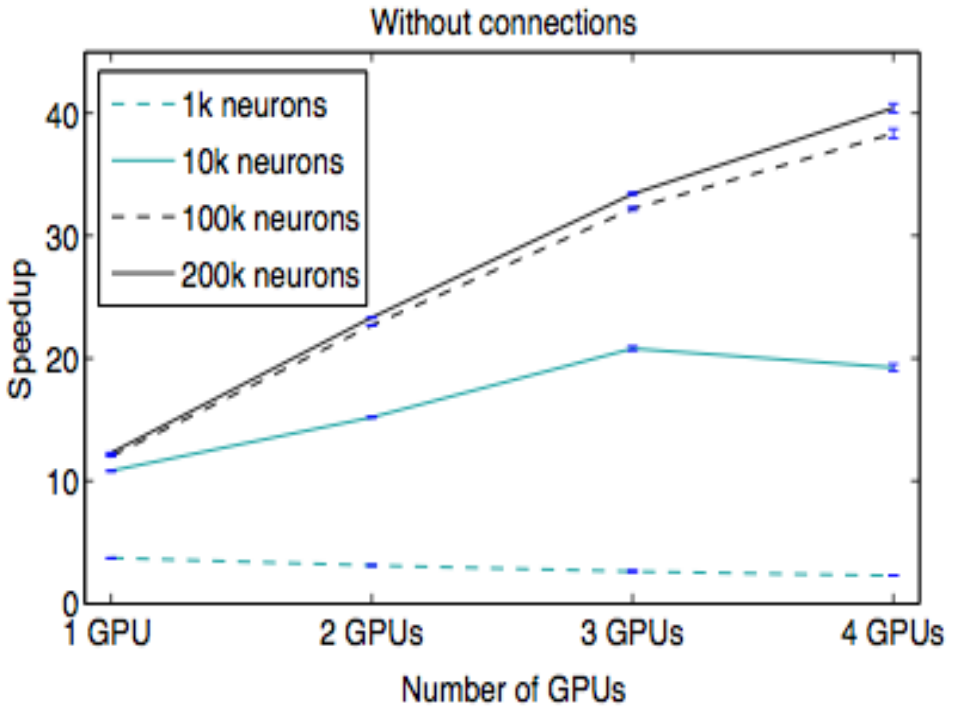


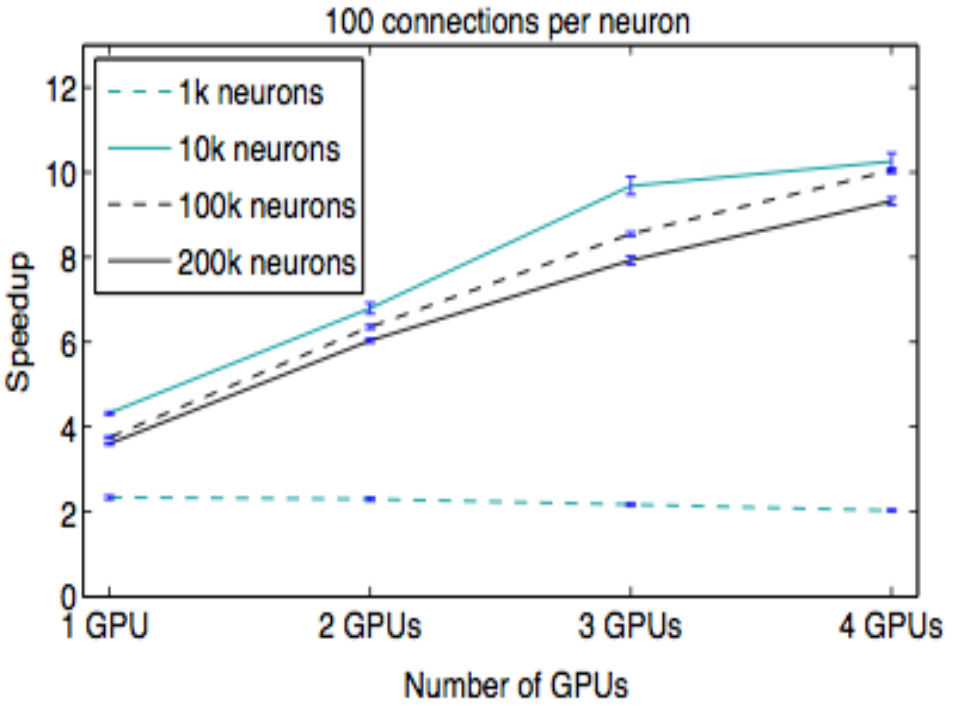


Resultados: Speedup

Sem conexões: ganho de 40x

Com 100 conexões / neurônio: ganho de 10x

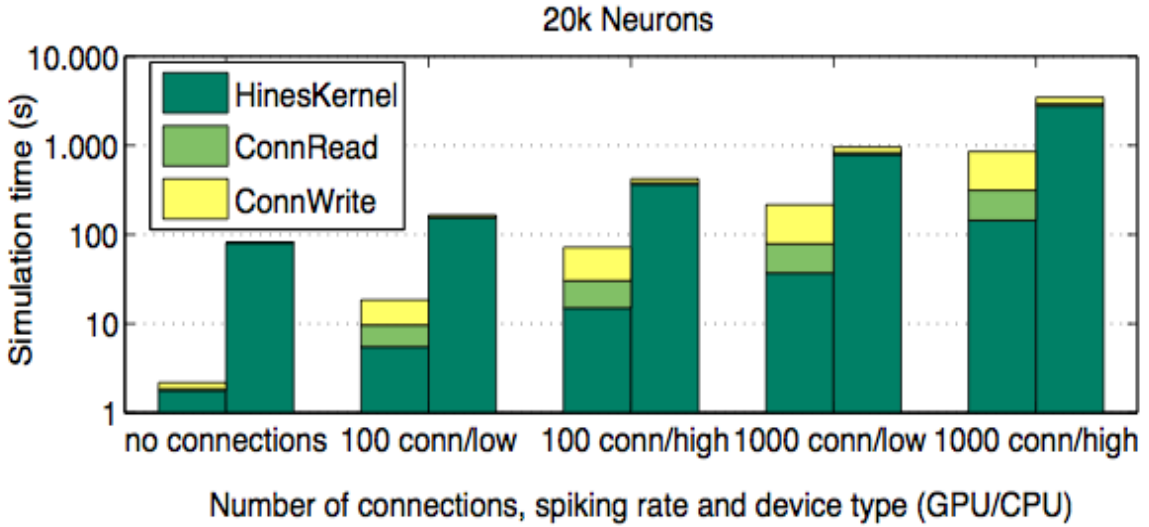


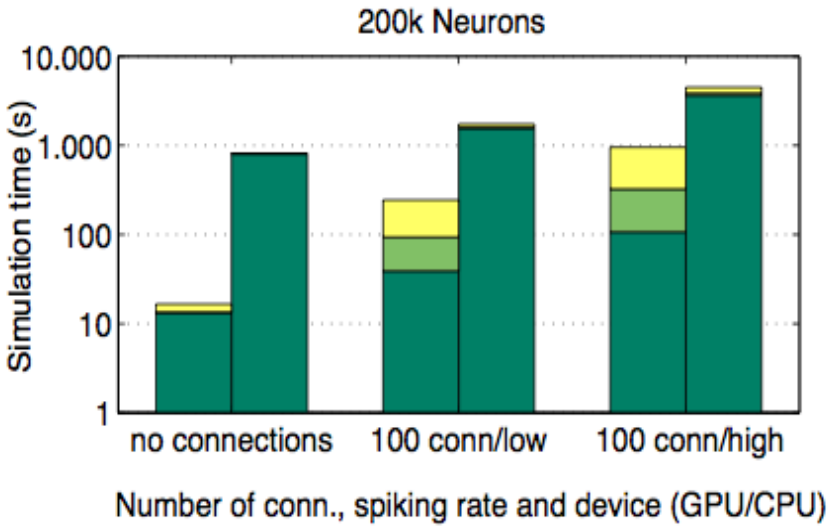


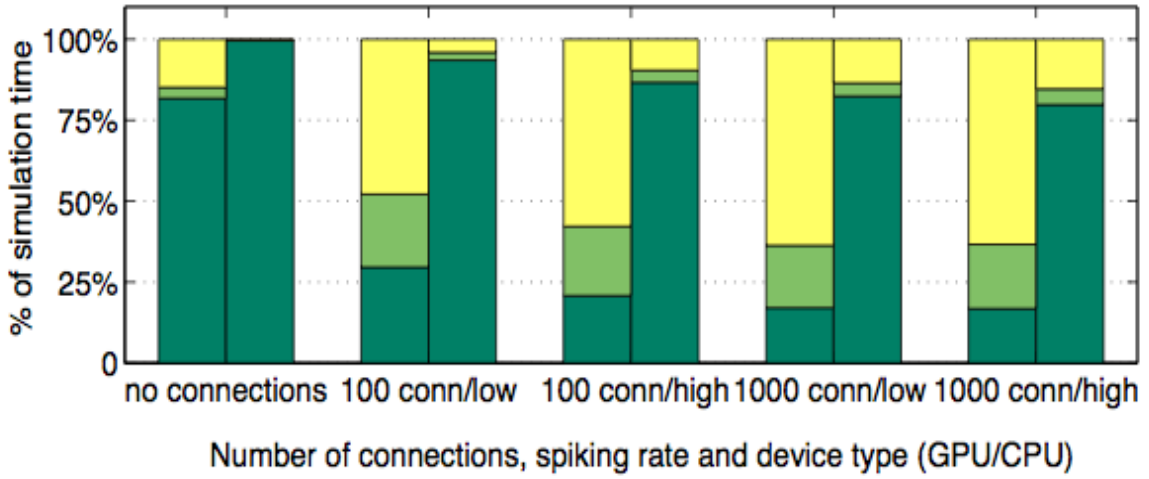
Resultados

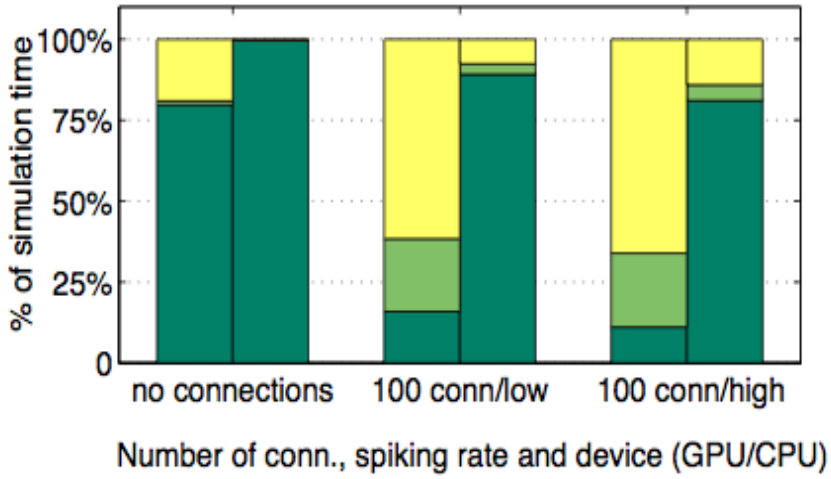
CPU: maior parte do tempo usada na integração numérica

GPU: maior parte do tempo é usada para comunicação









Estado da Arte em Simulações

Supercomputador IBM BlueGene
com 4096 processadores
Simulação de 4 milhões de
neurônios de 6 compartimentos
e 2 bilhões de sinapses

Brain-scale simulation of the neocortex on the IBM Blue Gene/L supercomputer

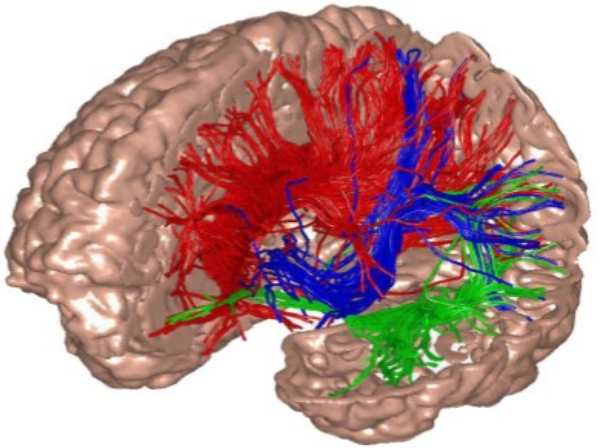
M. Djurfeldt
M. Lundqvist
C. Johansson
M. Rehn
Ö. Ekeberg
A. Lansner

Cluster com 60 processadores de
3GHz and 1.5 GB of RAM

Simulação de 1 milhão de neurônios
e meio bilhão de sinapses.
→ 10 minutos para inicializar e
1 minuto para simular 1 segundo
de funcionamento da rede

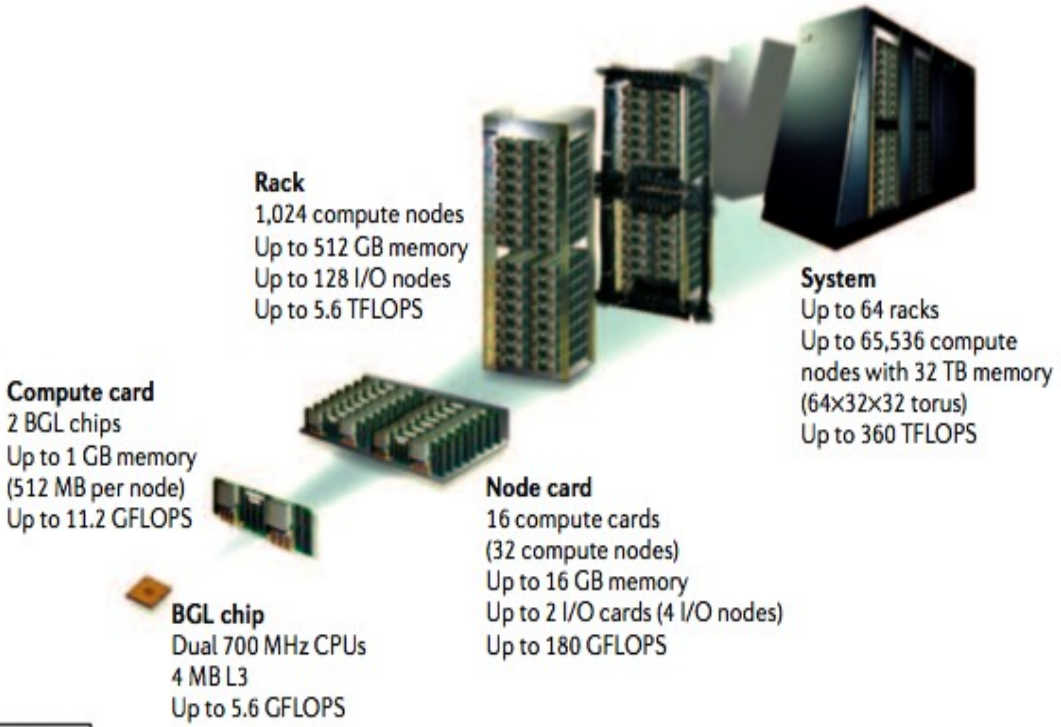
Large-scale model of mammalian thalamocortical systems

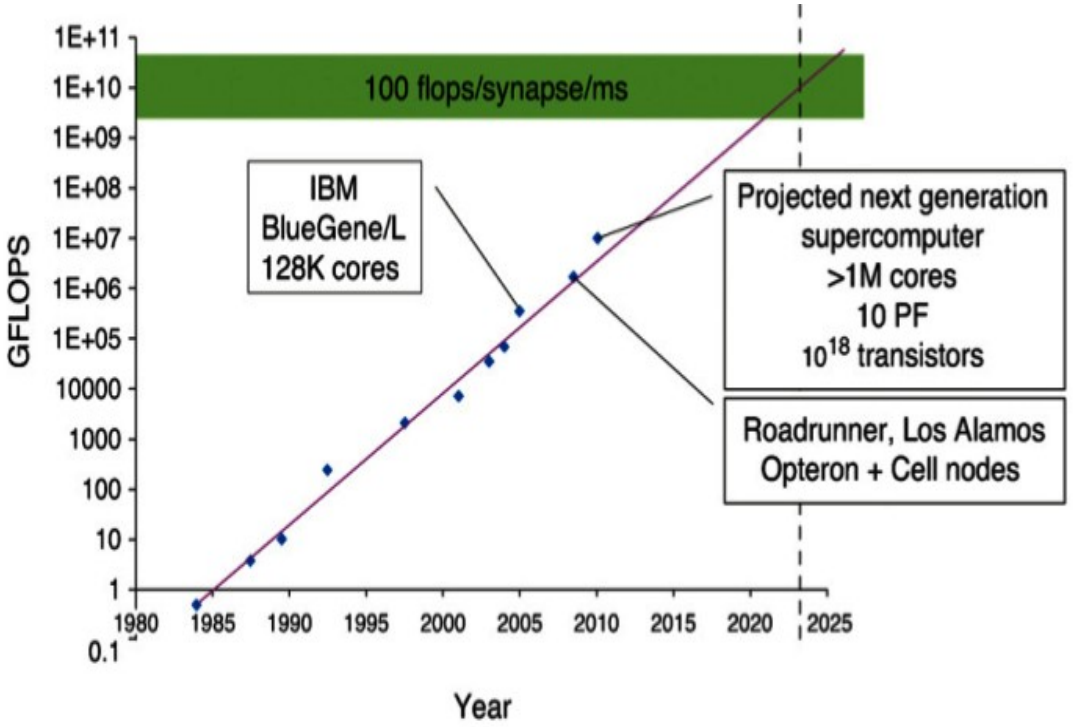
Eugene M. Izhikevich and Gerald M. Edelman*



Modelos de Grande Escala

Já estão sendo realizadas
simulações de modelos com
22 milhões de neurônios e
11 bilhões de sinapse





Supercomputadores
com dezenas de
milhares de
processadores

Parte V

Conclusões

Conclusões

Vimos que as GPUs permitem a obtenção de um excelente desempenho a um baixo custo

Composta por centenas processadores simples e um barramento compartilhado para acesso a memória

Mas:

Não fornecem bom desempenho para qualquer tipo de aplicação

Conclusões

A plataforma CUDA permite o uso de GPUs para executar aplicações paralelas

Extensão da linguagem C, sendo de fácil aprendizado

Mas para rodar de modo eficiente, é essencial que o código seja otimizado!

A linguagem OpenCL permite criar programas que rodam em múltiplas arquiteturas

Mas é difícil fazer um programa genérico que seja eficiente em arquiteturas diferentes

Onde Aprender Mais

Existem uma grande quantidade de material na Internet sobre CUDA. No site da nVidia existe links para diversos tutoriais e cursos online sobre a arquitetura CUDA

http://www.nvidia.com/object/cuda_education.html

- Para quem domina o inglês falado, neste site tem um curso em vídeo sobre CUDA dado por um engenheiro da nVidia na Universidade de Illinois

- Outra opção é um curso online (texto) publicado no site Dr. Dobbs.

- A distribuição do CUDA vem com 2 guias:
  CUDA Programming Guide e CUDA Best Practices