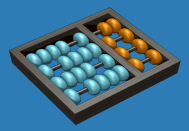
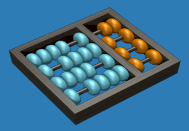


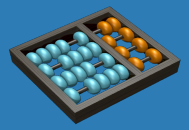
Arquitetura e Programação de GPU

- Leandro Zanotto – RA: 001962
- Anselmo Ferreira – RA: 023169
- Marcelo Matsumoto – RA: 085973



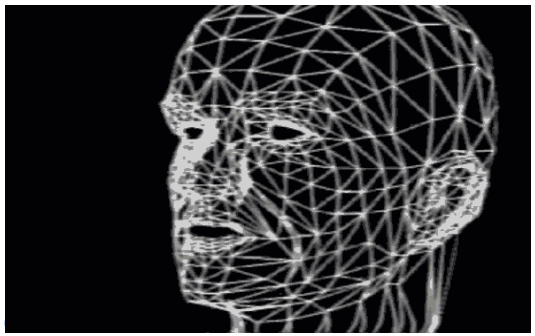
Agenda

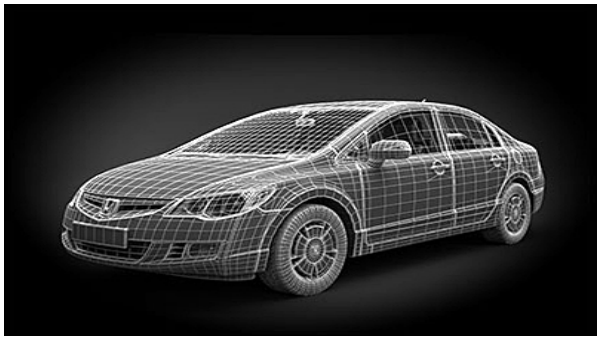
- Primeiras Placas de Vídeo
- Primeira GPU
- Arquitetura da GPU NVIDIA
- Arquitetura FERMI
- Arquitetura KEPLER
- Programação com CUDA



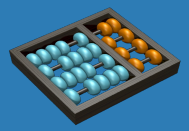
Primeira Geração – Wireframe

- Vértices: transformação, recorte e projeto
- Rasterização: somente linhas
- Pixel: sem pixels!
- Framebuffers com poucos bit por pixel em uma pequena resolução
- Datas: antes de 1987





UNICAMP



Segunda Geração – Sólidos Sombreados

- Vértice: iluminação
- Rasterização: polígonos preenchidos
- Pixel: buffer maior, mistura e cores
- Datas: 1987 -1992

Terceira Geração – Mapeamento de Texturas

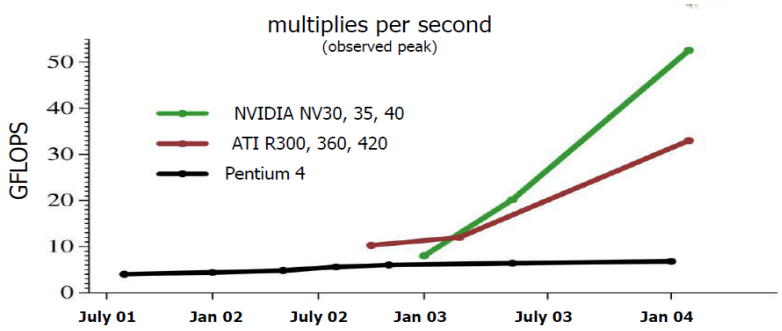
- Vértices e Rasterização: mais e mais rápidos
- Pixel: filtro de texturas, antialiasing
- Datas:1992 -2001

UNICAMP

História da GPU

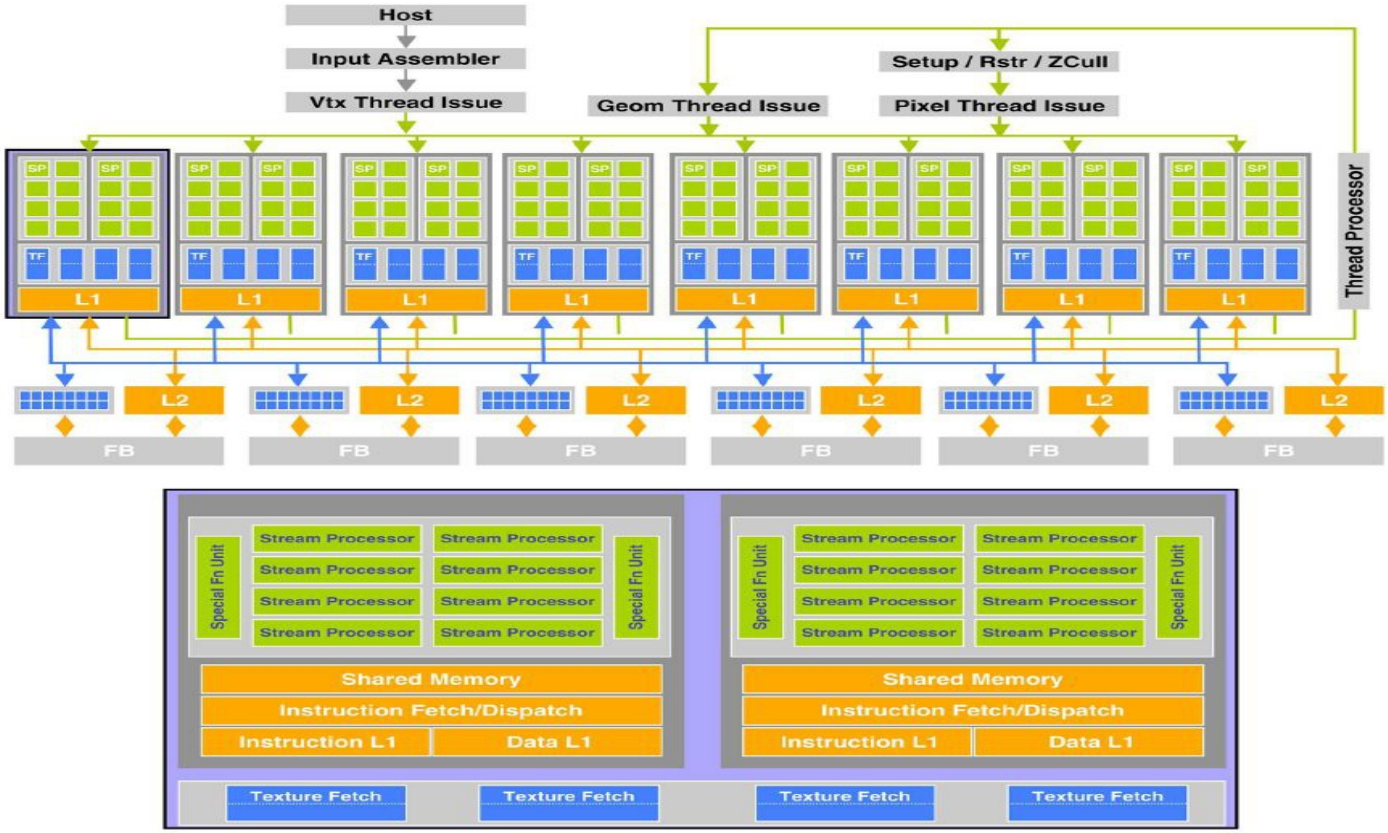
	Produto	Processo	Transistores	MHz	GFLOPS (MUL)
Aug-02	GeForce FX5800	0.13	**121M**	**500**	**8**
Jan-03	GeForce FX5900	0.13	**130M**	**475**	**20**
Dec-03	GeForce 6800	0.13	**222M**	**400**	**53**

- As GPUs já estavam com bom poder de processamento, mas ainda não eram programáveis
- Primeira GPGPU foi lançada em 2002

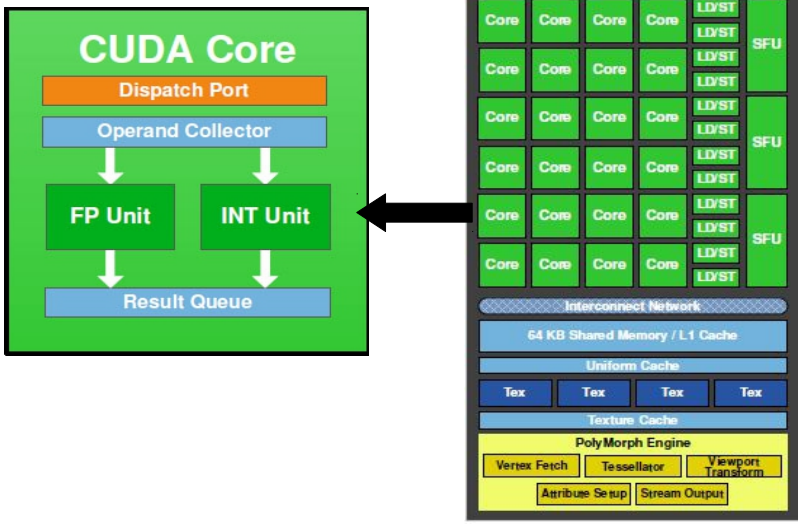


UNICAMP

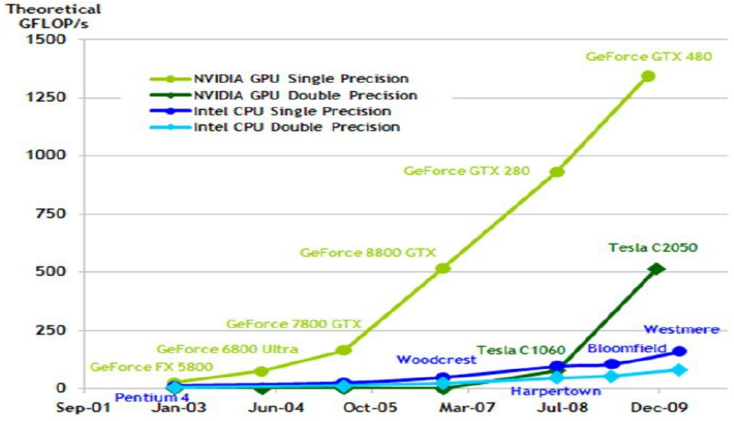
Arquitetura de uma GPU

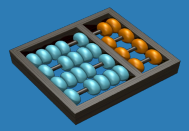


Arquitetura de GPU (cont...)



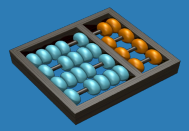
- CUDA Core (Stream Processor) e uma unidade programável
- Não existe Coerência de Cache L1
- Os CUDA Cores são organizados em blocos, cada bloco tem muitas threads.
- Os blocos podem ser organizados em grids.
- As threads podem ser sincronizadas
- A progamação é altamente paralela
- O desempenho é grande comparado com as CPUs atuais





Arquitetura Fermi

- Melhoria do desempenho na precisão dupla
- Suporte a ECC
- Hierarquia de Memória Verdadeira
- Maior memória compartilhada
- Troca de contexto mais rápida
- Operações Atômicas mais Rápidas



Arquitetura Fermi (cont)...

Terceira Geração de Streaming Multiprocessor (SM)

- 32 CUDA cores por SM, 4x sobre GT200
- 8x o pico de precisão dupla em ponto flutuante sobre GT200
- Warp Scheduler duplo, schedules simultâneos e instruções de despacho

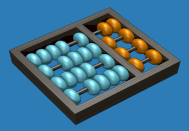
de dois warps independentes

- 64 KB de RAM com particionamento configurável da memória compartilhada e cache L1

Segunda Geração de execução de threads paralelas ISA

- Espaço de endereço unificado com suporte completo a C++
- Otimizado para OpenCL e DirectCompute
- Total IEEE 754-2008 de precisão de 32-bit e 64-bit
- Caminho inteiro total de 32 bits com extensões de 64-bits
- Acesso às instruções de memória e suporte de transição a endereços de 64-bit
- Melhoria do desempenho através de predição

UNICAMP



Arquitetura Fermi

Melhorada a memória do subsistema

- Hierarquia NVIDIA Parallel DataCacheTM com L1 configurável e L2 unificado

Caches

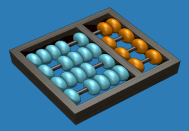
- Primeira GPU com ECC
- Melhoria no desempenho das operações atômicas

NVIDIA GigaThreadTM Engine

- Troca de contexto 10x mais rápida
- Execução de kernel concorrente
- Execução do bloco de thread fora de ordem
- Transferência de memória dupla

Arquitetura Kepler

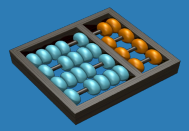
Streaming Multiprocessor (SMX) Architecture



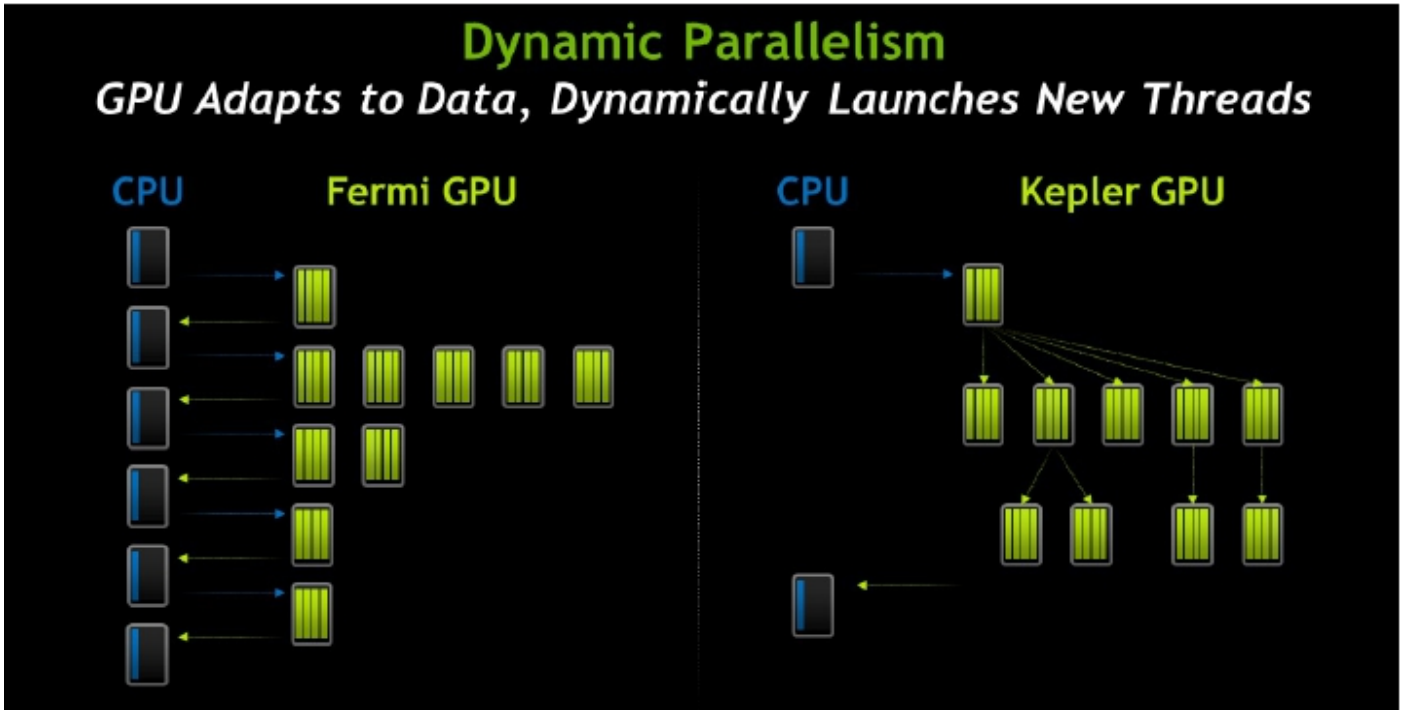
Arquitetura Kepler

- Quad Warp Scheduler
- Performance per Watt
- Dynamic Parallelism
- Hyper-Q
- Grid Management Unit
- NVIDIA GPUDirect™

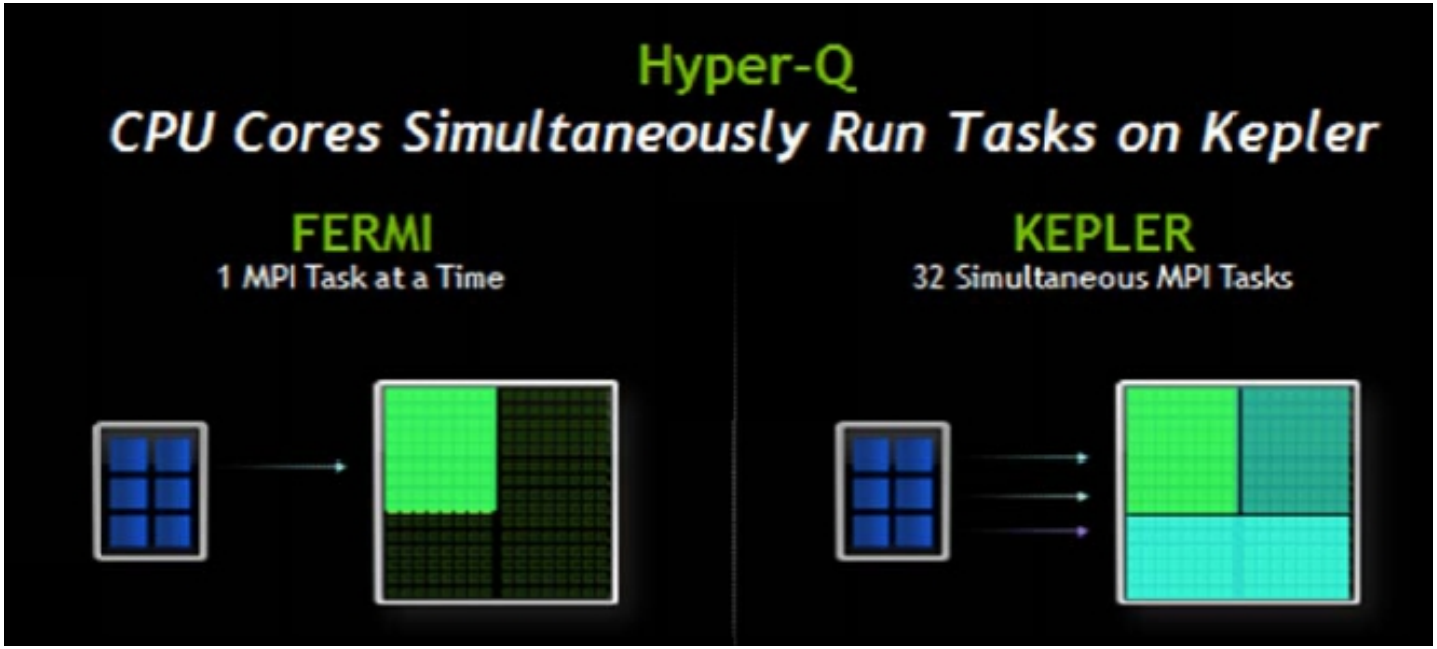
UNICAMP



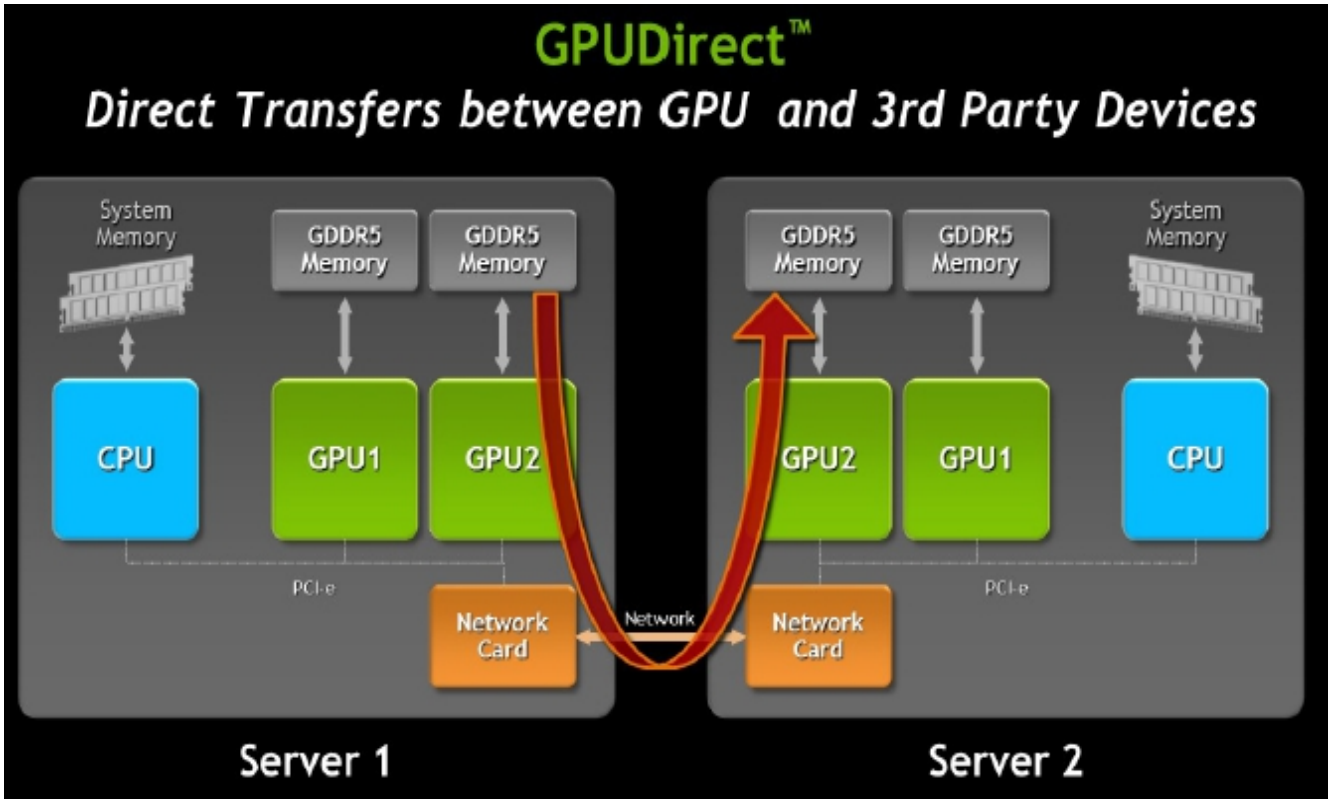
Arquitetura Kepler

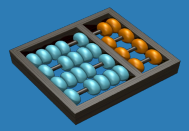


Arquitetura Kepler



Arquitetura Kepler



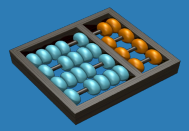


Arquitetura Kepler

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

UNICAMP

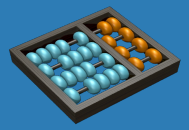


O que é CUDA?

Arquitetura do CUDA (Compute Unified Device Architecture)

- Expõe à GPU a capacidade de ser programada para propósitos gerais
- Retém o desempenho dos tradicionais DirectX/OpenGL
- CUDA C
- Baseado nos padrões do C
- Uma linguagem útil para as extensões que permitem fazer
- programas heterogêneos
- API direta para gerenciar dispositivos, memória, etc.
- Só funciona com as GPUs da nVidia
- Existe no mercado o OpenCL

UNICAMP



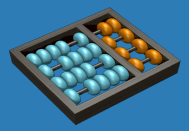
CUDA - Terminologia

- Host – A CPU e sua memória (memória do host)
- Device – A GPU e sua memória (memória do device)

Host

Device

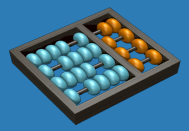
UNICAMP



Hello, World!

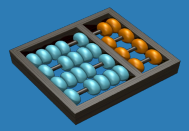
```
int main( void) {
printf( "Hello, World!\n" );
return 0;
}
```

- Este programa simples em C é executado no host
- O compilador da nVidia (nvcc) não irá reclamar se o programa não tiver código no device
- CUDA é simples como C



Hello, World! no device

```
__global__ void kernel( void) {
}
int main( void) {
kernel<<<1,1>>>();
printf( "Exemplo 1!\n" );
return 0;
}
```



Hello, World! no device

```
__global__ void kernel( void) {
}
```

A palvra chave do CUDA __global__ indica que a função

- É executada no device
- É chamado do host

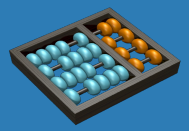
nvcc divide o código fonte em componentes de host e device

- o nvcc manipula as funções do device como kernel()
- o compilador padrão do host manipula funções como main()

Pode-se usar:

- gcc para Linux
- Microsoft Visual C++

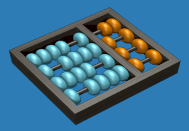
UNICAMP



Hello, World! Com o código no device

```
int main( void) {
kernel<<< 1, 1 >>>();
printf( "Hello, World!\n" );
return 0;
}
```

Os <<< >>> são chamadas de kernel através do host



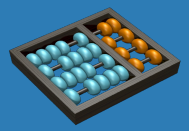
Um outro exemplo

Um kernel que adiciona dois inteiros

```
__global__ void add( int*a, int*b, int*c ) {
*c = *a + *b;
}
```

__global__ é um palavra chave de CUDA que siginfica:

- add() será executado no device
- add() será chamado pelo host

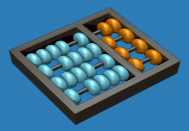


Um outro exemplo

Perceba que usamos ponteiros em nossas variáveis

```
__global__ void add( int*a, int*b, int*c ) {
*c = *a + *b;
}
```

Add() é executado no device, então a,b e c devem apontar para a memória do device



Gerenciamento de Memória

A memória do Host e do device são entidades distintas

Ponteiros do Device apontam para a memória da GPU

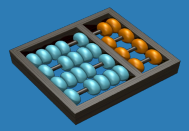
- Pode ser passado do código do host

Ponteiros do Host apontam para a memória da CPU

- Pode ser passado para o código do dispositivo

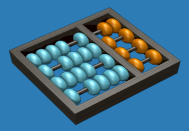
Comandos básicos para manipular a memória com CUDA

- **cudaMalloc(), cudaFree(), cudamemcpy()**
- Similar em C temos **malloc(), free(), memcpy()**



Exemplo com gerenciamento de memória

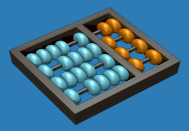
```
int main( void ) {
int a, b, c; // host copia de a, b, c
int*dev_a, *dev_b, *dev_c; // device copia de a, b, c
int size = sizeof( int); // precisamos de um espaço para um inteiro
// aloca no device cópias de a, b, c
cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
a = 2;
b = 7;
```



Exemplo com gerenciamento de memória (cont)

```
// copia a entrada para o device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);
// executa add() kernel na GPU, passando parâmetros
add<<< 1, 1 >>>( dev_a, dev_b, dev_c);
// copia o resultado do device de volta para o host a cópia de c
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
```

UNICAMP



Programação Paralela com CUDA

A computação de GPU é a massiva paralela

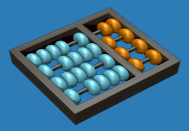
Para executarmos o programa anterior em paralelo temos que fazer uma pequena mudança

add<<< **1, 1** >>>(dev_a, dev_b, dev_c);

add<<< **N, 1** >>>(dev_a, dev_b, dev_c);

Feito isso o add() será executado N vezes em paralelo

UNICAMP



Programação Paralela com CUDA

Cada invocação do add() se refere como um bloco

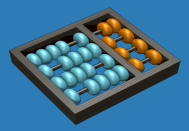
Kernel pode se referir ao índice do bloco com a variável blockIdx.x

Para a soma de um vetor, cada bloco adiciona o valor de a[] e b[] tendo seu resultado em c[]:

```
__global__ void add( int*a, int*b, int*c )
{
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Utilizando blockIdx.x para os índices dos arrays, cada bloco manipula índices diferentes

UNICAMP



Programação Paralela com CUDA

O código anterior:

```
__global__ void add( int*a, int*b, int*c )
{
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Este código será executado no device assim:

Bloco 0

```
c[0] = a[0] + b[0];
```

Bloco 1

```
c[1] = a[1] + b[1];
```

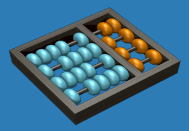
Bloco 2

```
c[2] = a[2] + b[2];
```

Bloco 3

```
c[3] = a[3] + b[3];
```

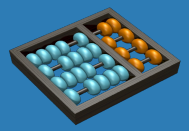
UNICAMP



Adição em Paralelo: add()

Usando nos kernel paralelizado add():

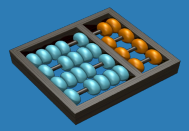
```
__global__ void add( int*a, int*b, int*c )
{
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```



Adição em Paralelo: main()

```
#define N 512
int main( void ) {
int a, b, c; // cópias de a,b, c no host
int*dev_a, *dev_b, *dev_c; // cópias de a,b, c no device
int size = N * sizeof( int); // precisa-se do espaço para N inteiros
// aloca a cópias de a, b, c no device
cudaMalloc( (void**)&dev_a, size ); cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
a = 2; b = 7;
// copia a entrada para o device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);
// executa o add() kernel na GPU, passando parametros
add<<< N, 1 >>>( dev_a, dev_b, dev_c);
// copia o resultado do device de volta para o host com a cópia e c e libera a memória alocada na GPU
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);
cudaFree( dev_a); cudaFree( dev_b); cudaFree( dev_c);
return 0; }
```

UNICAMP



Threads

Um bloco pode ser dividido em **threads** paralelas

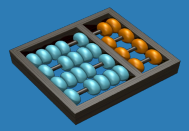
Mudando a adição do vetor para usar threads paralelas:

Bloco

```
__global__ void add( int*a, int*b, int*c )
{
c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

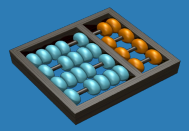
Thread

```
__global__ void add( int*a, int*b, int*c ) {
c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```



Adição Paralela com Threads

```
#define N 512
int main( void ) {
int*a, *b, *c; // cópias de a,b, c no host
int*dev_a, *dev_b, *dev_c; // cópias de a,b, c no device
int size = N * sizeof( int); // precisa-se do espaço para 512 inteiros
// aloca a cópias de a, b, c no device
cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
a = (int*)malloc( size );
b = (int*)malloc( size );
c = (int*)malloc( size );
random_ints( a, N );
random_ints( b, N );
```



Adição Paralela com Threads (cont)

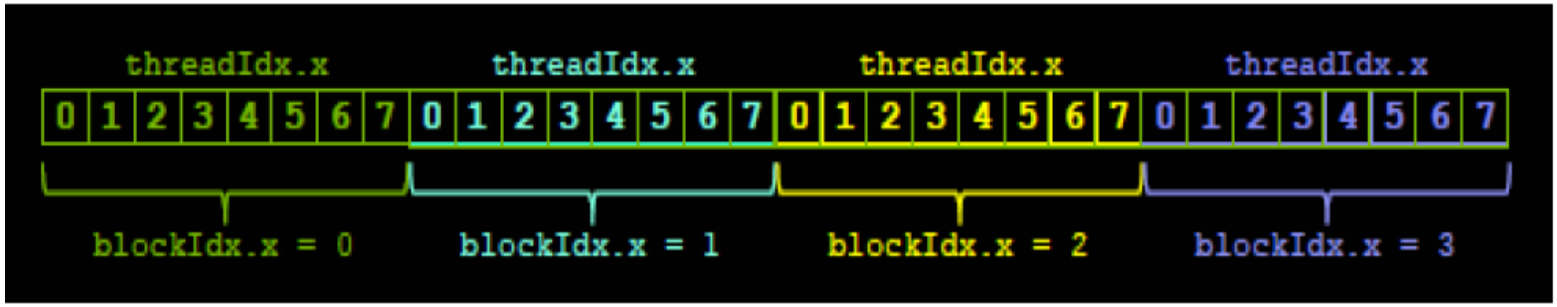
```
// copia a entrada para o device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);
// executa o add() kernel com N threads
add<<<N, N>>>( dev_a, dev_b, dev_c);
// copia o resultado do device para o host a cópia de c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);
free( a ); free( b ); free( c );
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
return 0;
}
```

UNICAMP

Indexando Arrays com Threads e Blocos

Não basta usar threadIdx.x ou blockIdx.x

Para indexar um array com 1 thread por entrada (8 threads por bloco)

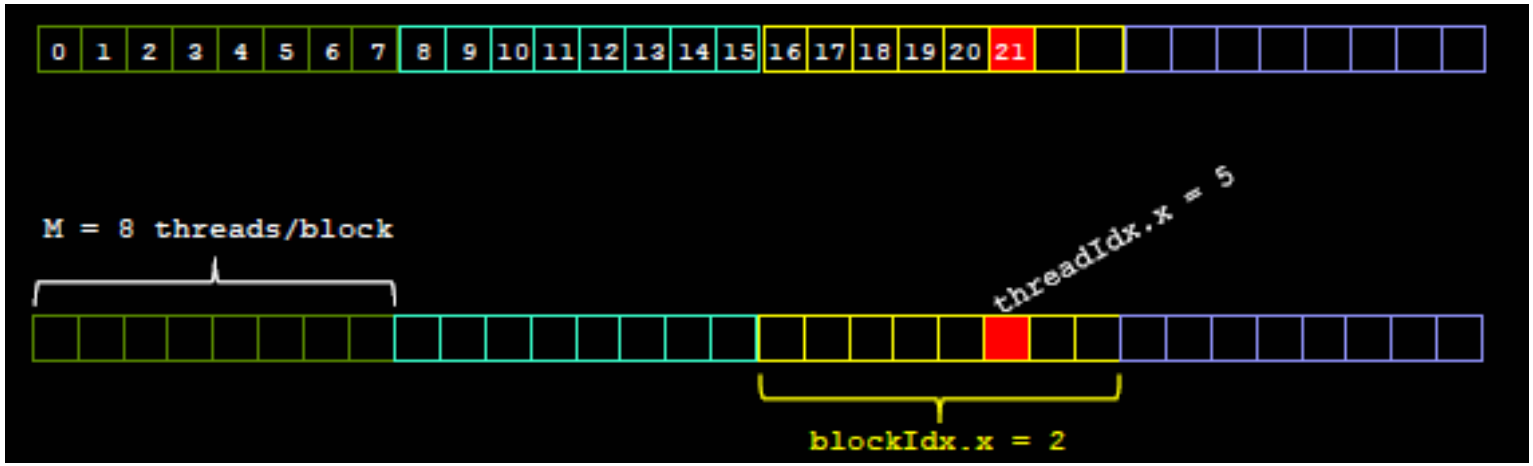


Se nós temos M threads/blocos, um único índice do array para cada entrada é dado por

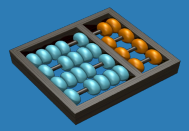
```
int index = threadIdx.x + blockIdx.x * M;

int index =       x     +      y      * width;
```

Indexando Arrays - Exemplo

Para a entrada em vermelho temos o índice 21:




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +      2      * 8;
          = 21;
```



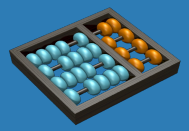
Adição com Threads e Blocos

Temos uma variável feita para utilizar threads / blocos (blockDim.x)

```
int index= threadIdx.x + blockIdx.x * blockDim.x;
```

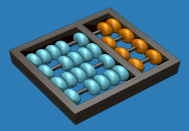
Uma versão da adição de vertor utilizando blocos e threads:

```
__global__ void add( int*a, int*b, int*c ) {
int index = threadIdx.x + blockIdx.x * blockDim.x;
c[index] = a[index] + b[index];
}
```



Adição Paralela com Threads e Blocos

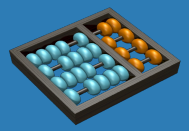
```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main( void ) {
int*a, *b, *c; // cópias de a,b, c no host
int*dev_a, *dev_b, *dev_c; // cópias de a,b, c no device
int size = N * sizeof( int); // precisa-se do espaço para N inteiros
// aloca a cópias de a, b, c no device
cudaMalloc( (void**)&dev_a, size );
cudaMalloc( (void**)&dev_b, size );
cudaMalloc( (void**)&dev_c, size );
a = (int*)malloc( size );
b = (int*)malloc( size );
c = (int*)malloc( size );
random_ints( a, N );
random_ints( b, N );
```



Adição Paralela com Threads e Blocos (cont)

```
// copia a entrada para o device
cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, b, size, cudaMemcpyHostToDevice);
// executa o kernel add() com blocos e threads
add<<< N/THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>( dev_a, dev_b,
dev_c);
// copia o resultado do device para o host com o a cópia de c
cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);
free( a ); free( b ); free( c );
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
Return 0;
}
```

UNICAMP



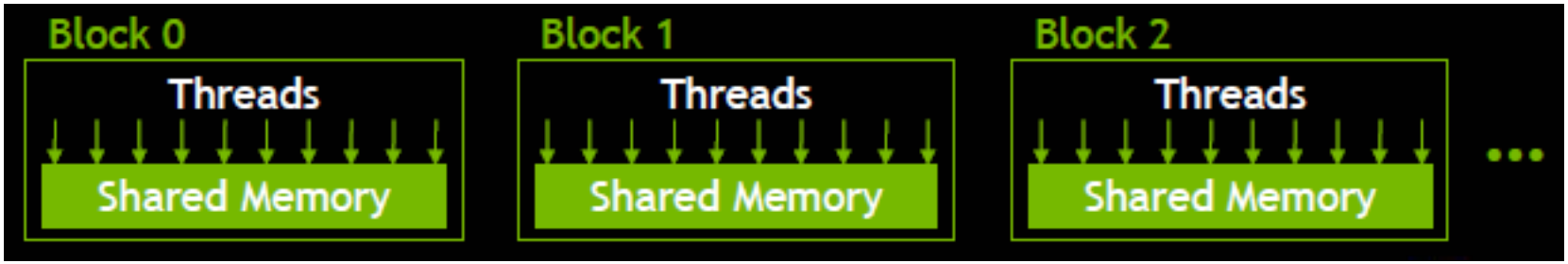
Compartilhando dados entre Threads

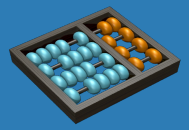
Um bloco de Threads compartilha a memória chamada de "***shared memory***"

Extremamente rápida, construída no próprio chip

Declarada com a palavra chave do CUDA __shared__

Somente visível para as threads do mesmo bloco





Sincronização de Threads

Precisamos que as threads aguardem entre as seções do Produto Escalar

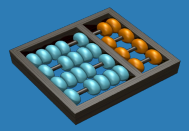
Sincronizamos as threads utilizando a função __syncthreads()

As threads do bloco esperam até que todas elas alcancem o __syncthreads()

As threads são somente sincronizadas com um bloco

```
__global__ void dot( int*a, int*b, int*c ) {
__shared__ int temp[N];
temp[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
__syncthreads(); // Aqui ocorre a sincronização das threads, elas vão
esperar até todas alcançarem este ponto
if( 0 == threadIdx.x ) {
int sum = 0;
for( int i= 0; i< N; i++ )
sum += temp[i];
*c = sum;
}
}
```

UNICAMP



Operações Atômicas

CUDA suporta operações atômicas como:

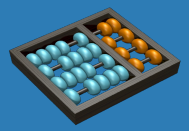
atomicAdd() atomicSub()

atomicMin() atomicMax()

atomicInc() atomicDec()

atomicExch() atomicCAS()

Utilizado para executar um read-modify-write sem interrupção



Perguntas