



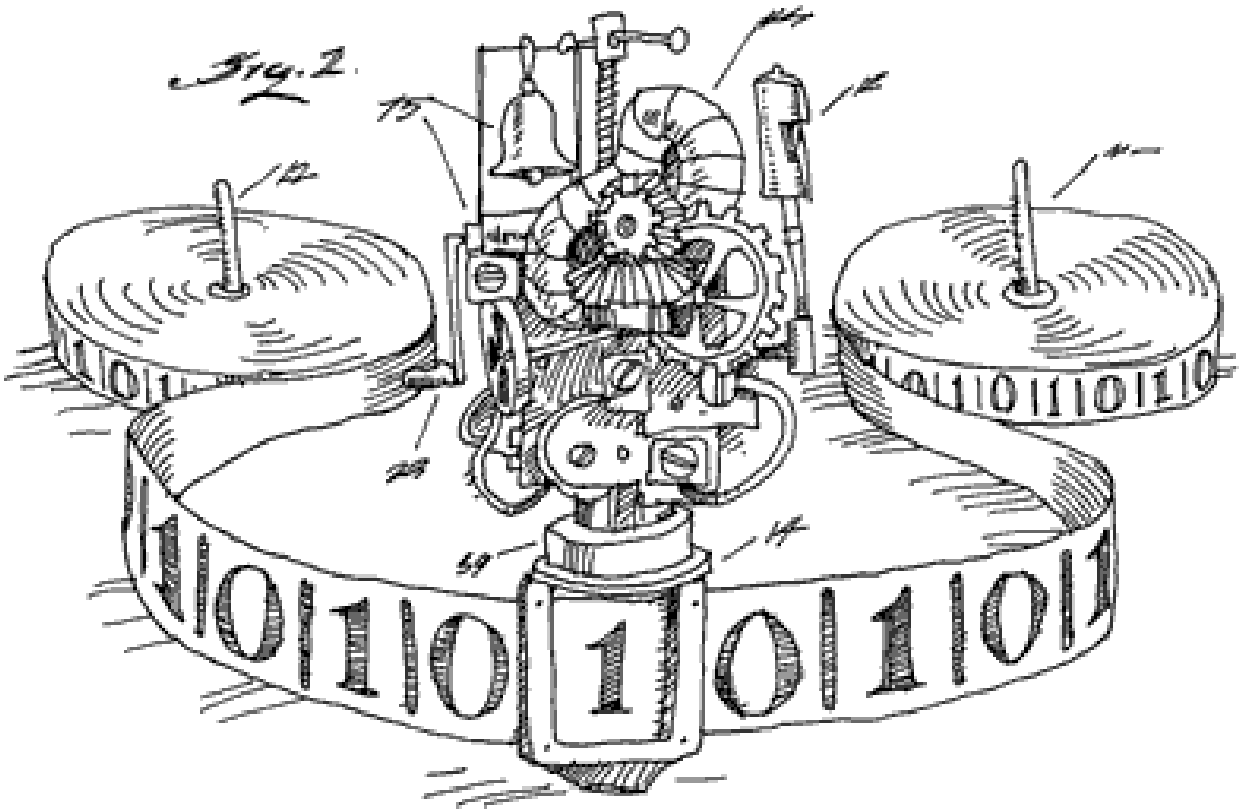
ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Computação em *GPU's*

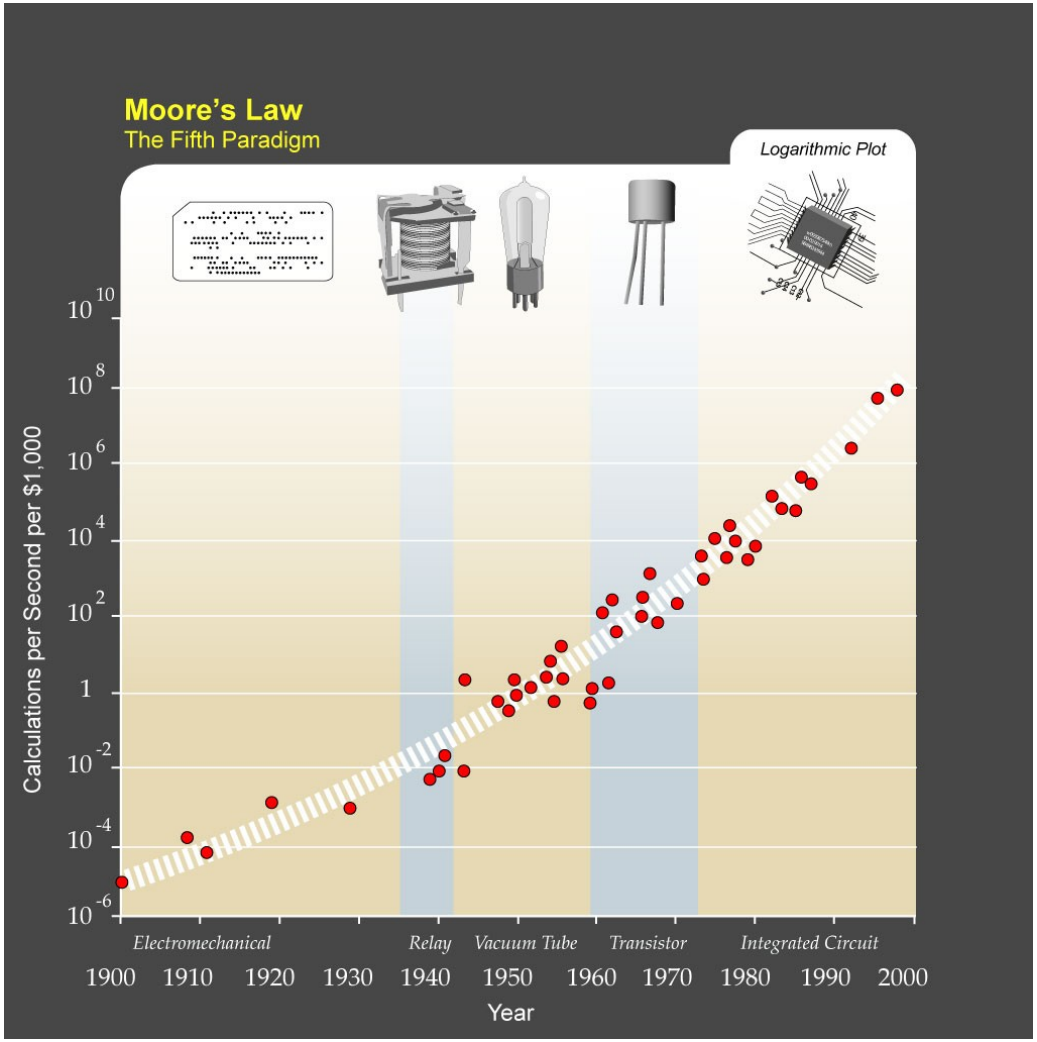
Prof. Thiago Martins

PMR5022

Computação Sequencial

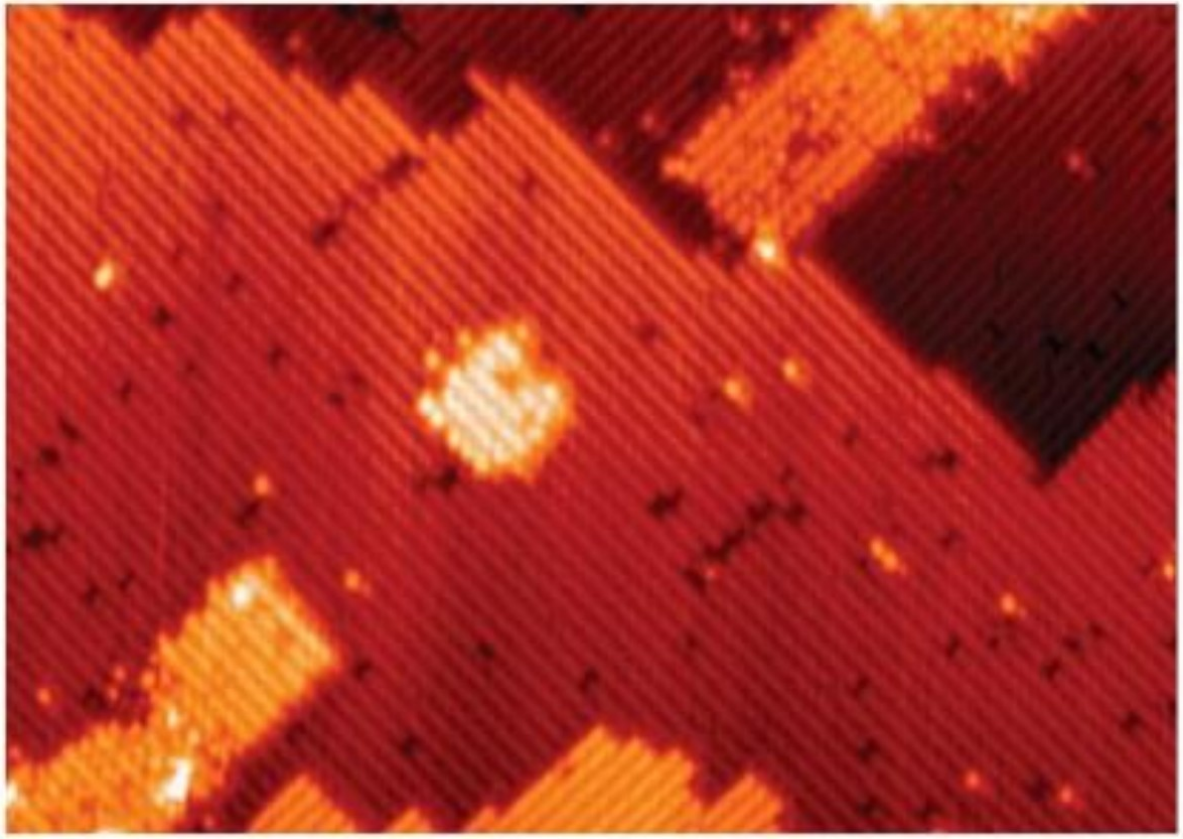

Fig. 2
1 0 1 1 0 1 1 0 1

Computação Sequencial



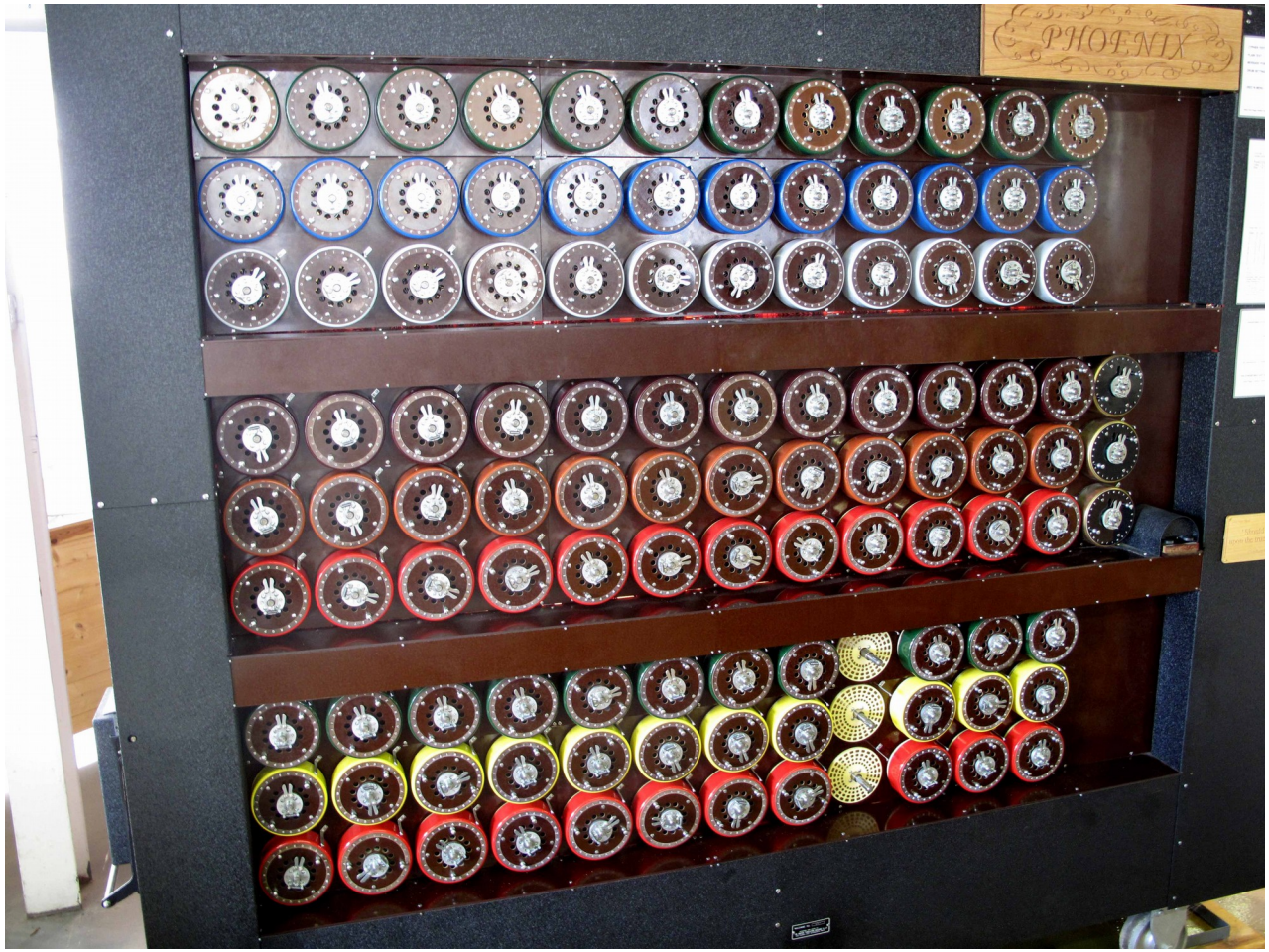
"Lei de Moore" - 1900-2000

Limites Físicos



Transistor de 4nm (apenas 7 átomos!) - 2010

Computação Paralela



Bomba Kryptologiczna – Versão de Bletchley Park - 1939

Computação Paralela

SIMD – Processamento vetorial



Cray 1- 1976

Computação Paralela

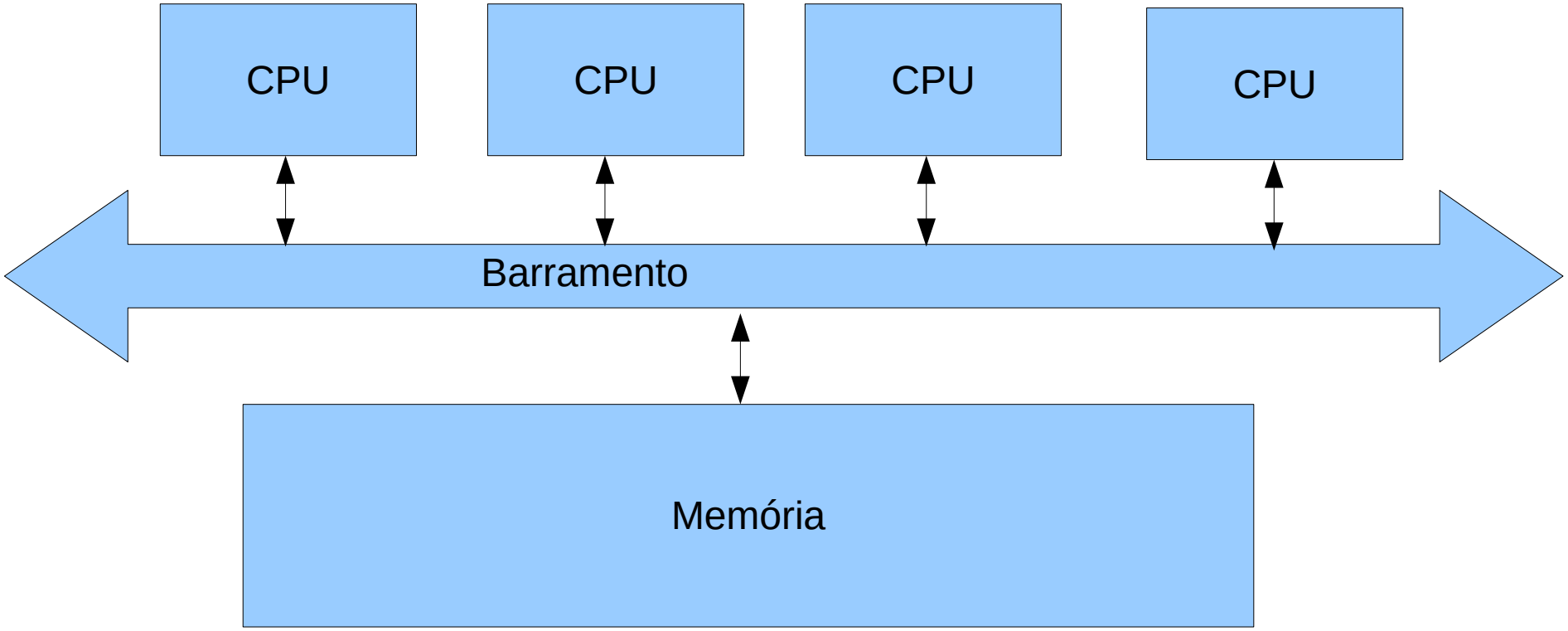
MIMD – Processamento paralelo massivo



Connection Machine CM5 1993

Computação Paralela

MIMD – Arquietura de memória uniforme

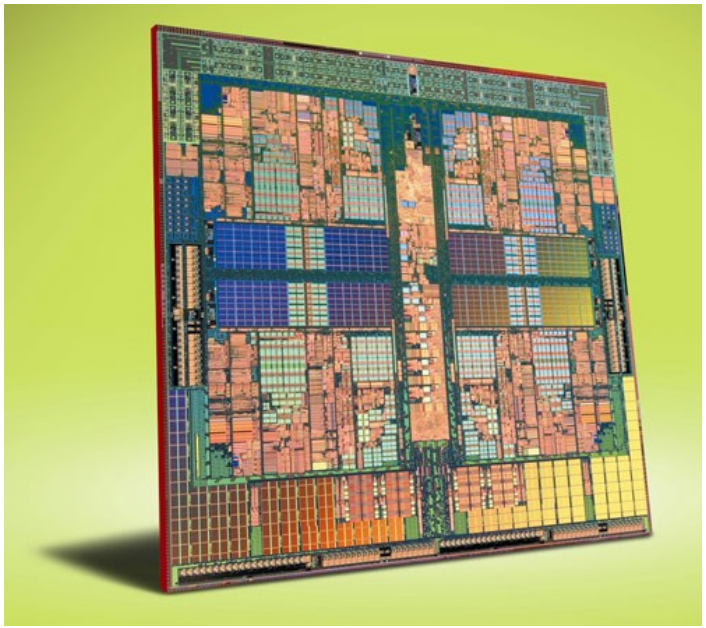


Computação Paralela

MIMD – Arquietura de memória uniforme



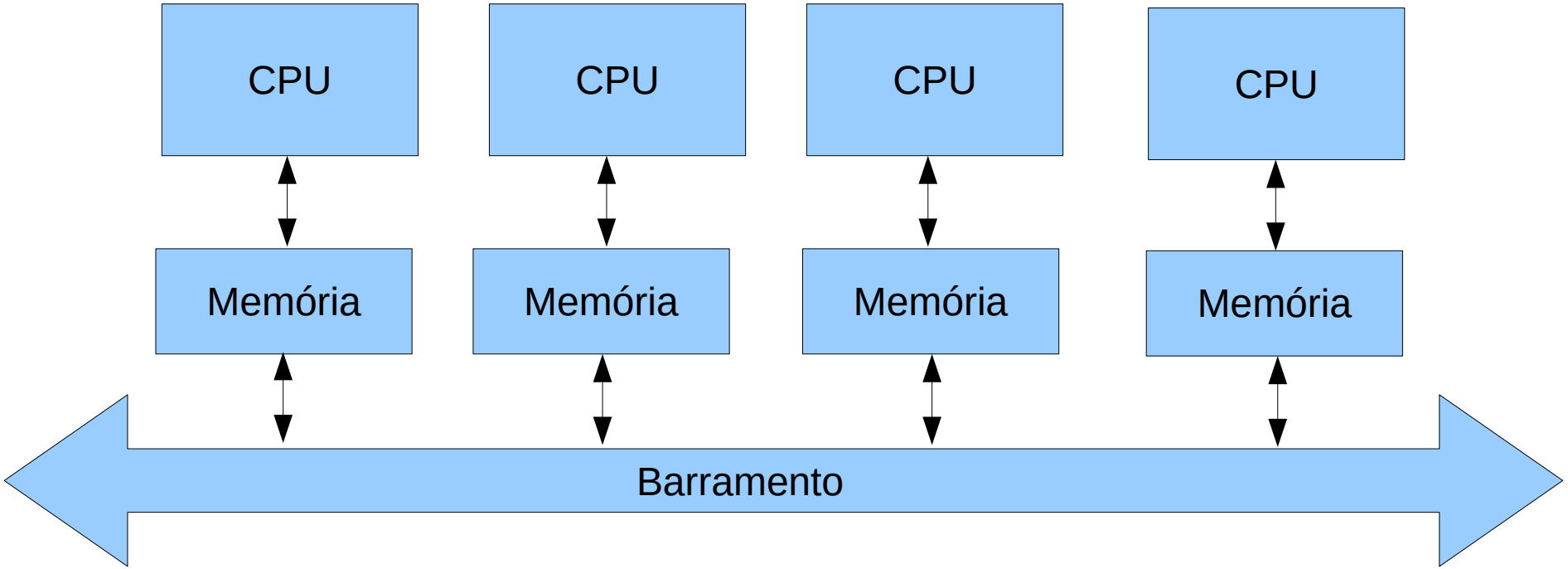
Motherboard quad Xeon - 1998



AMD Phenom - 2007

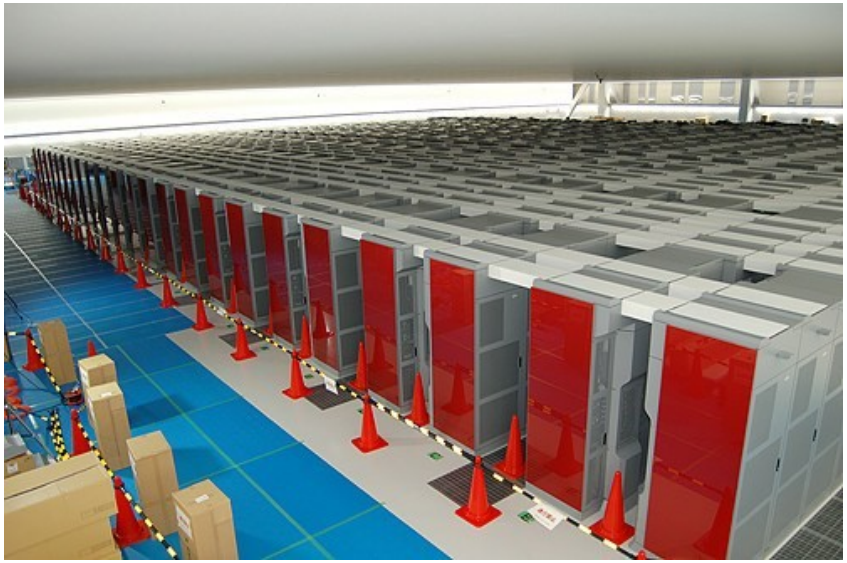
Computação Paralela

MIMD – Arquietura de memória náo-uniforme

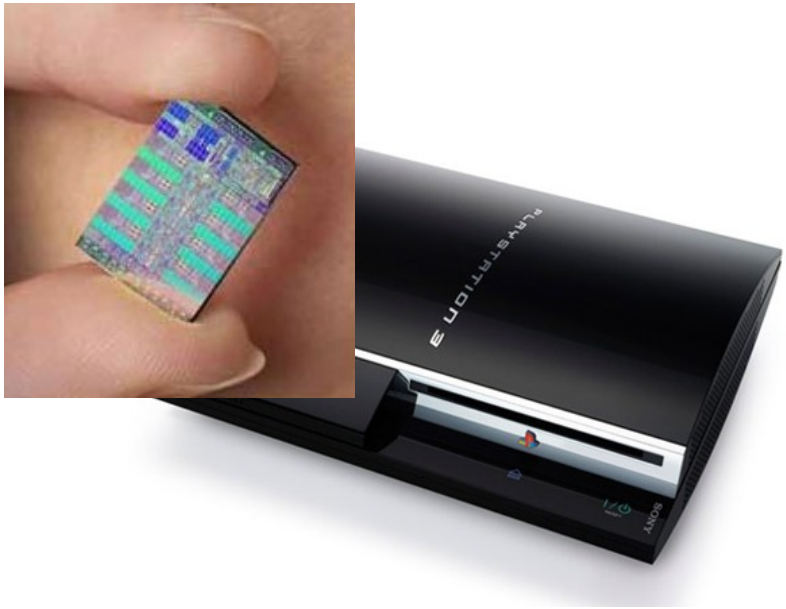


Computação Paralela

MIMD – Arquietura de memória não-uniforme

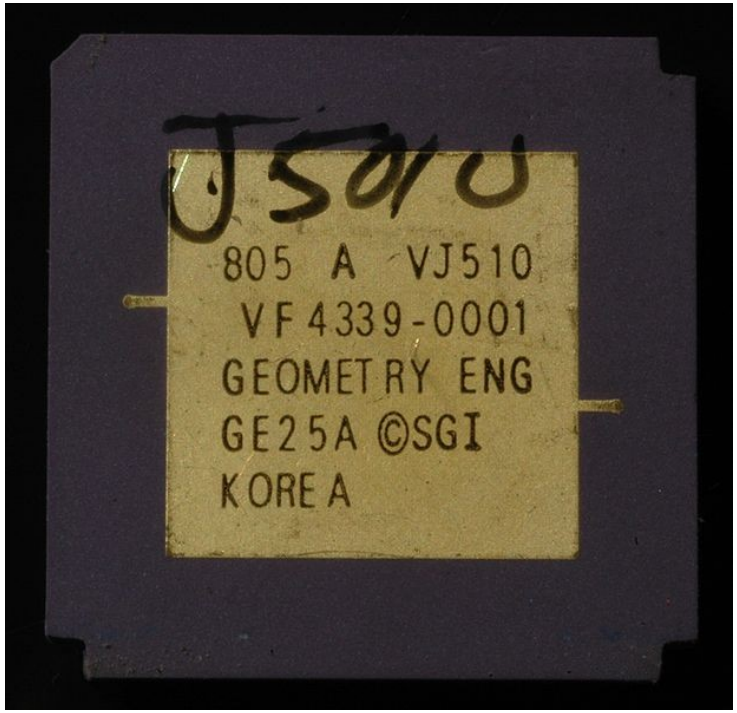


K-Computer - 2011



Cell Processor - 2005

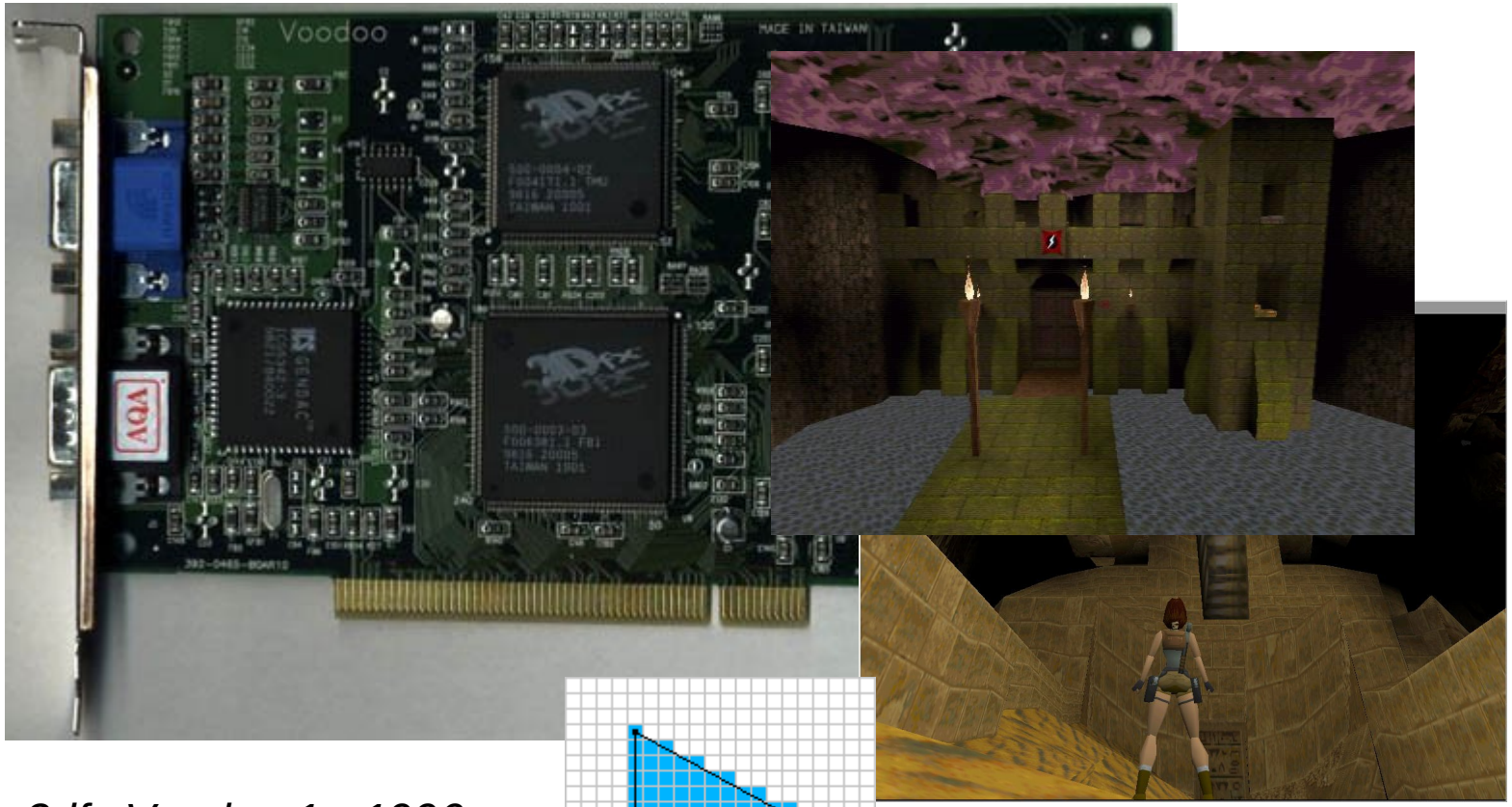
GPU's



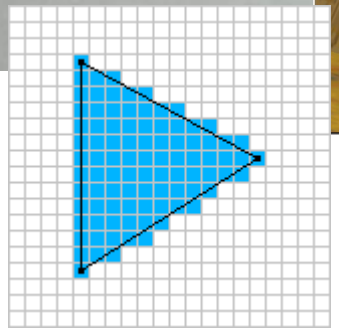
SGI Geometry Engine - 1989



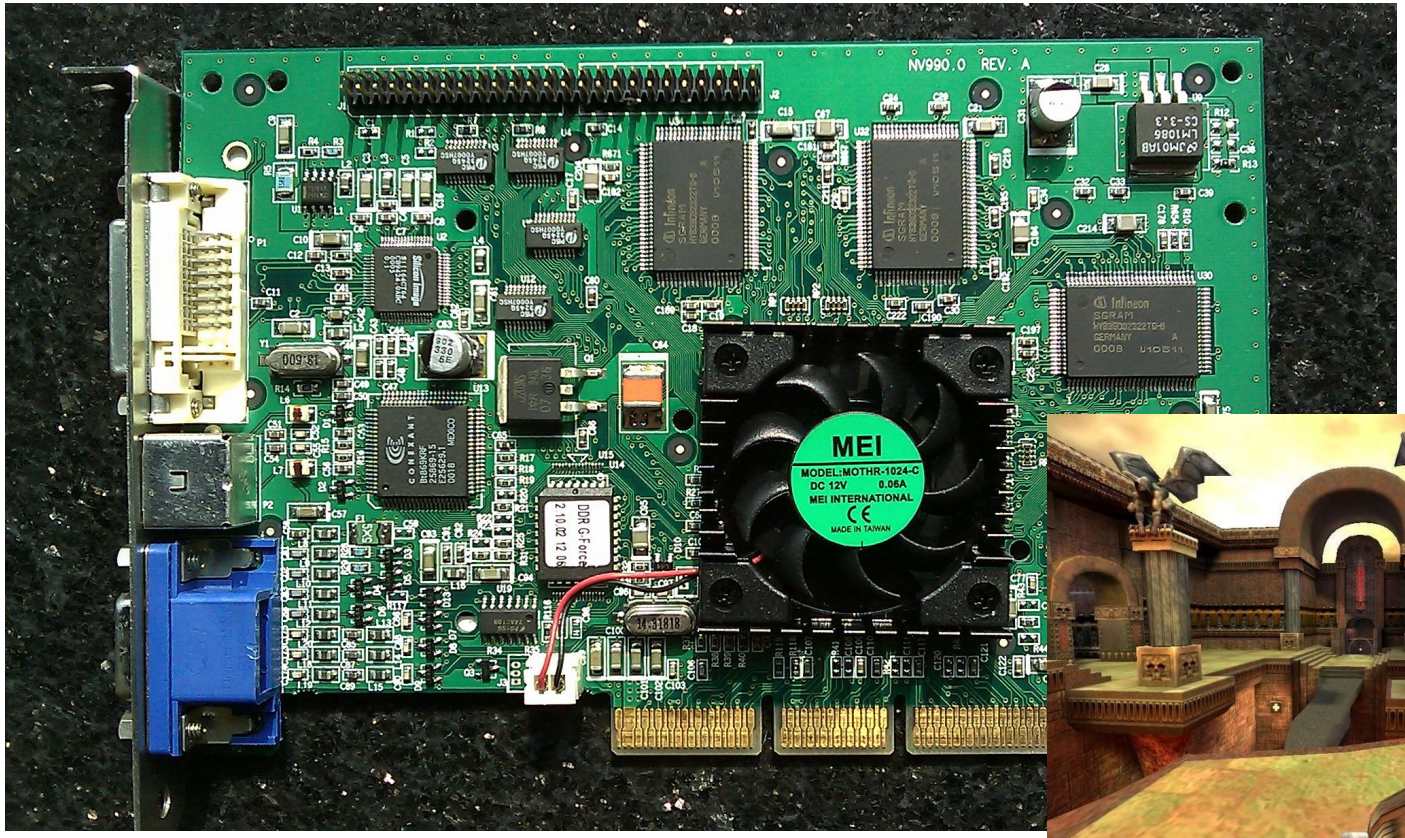
GPU's



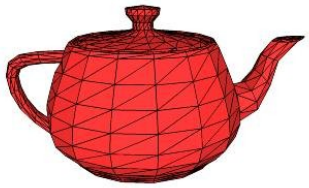
3dfx Voodoo 1 - 1996



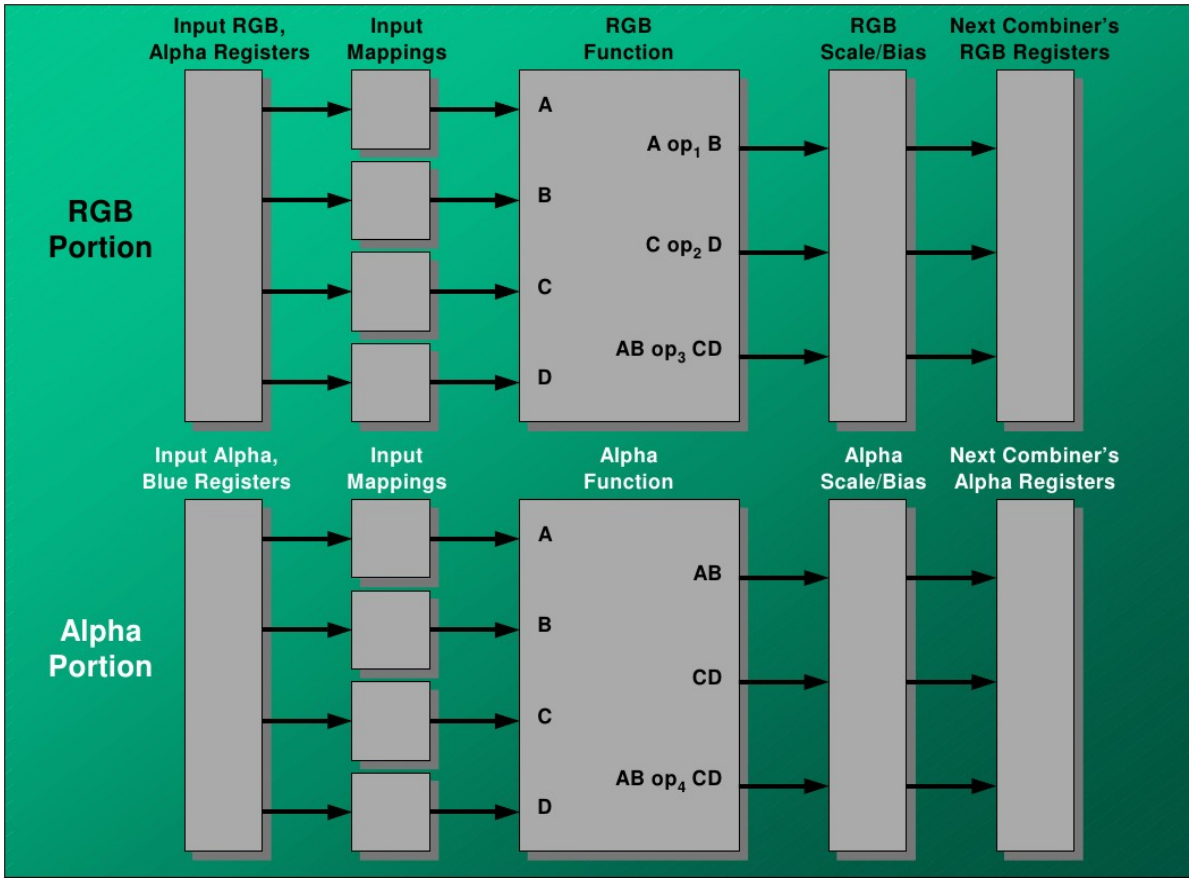
GPU's



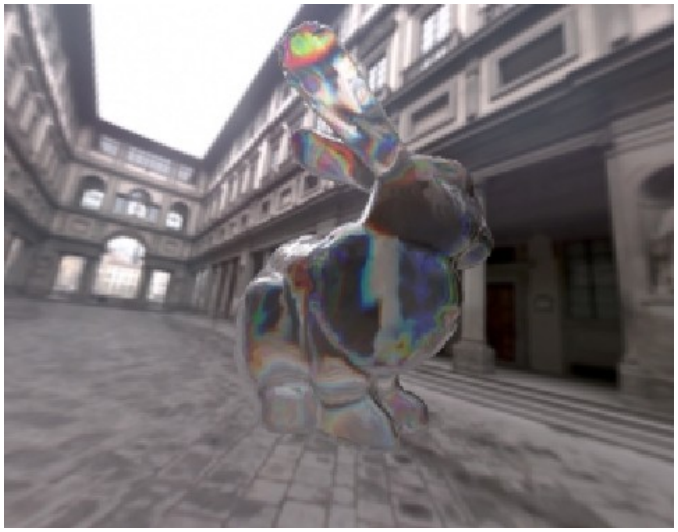
NVIDIA GeForce 256 - 1999



GPU's



NVIDIA GeForce 256 – Combinador de texturas
(reproduzido de "Texture Compositing With Register Combiners" - John Spitzer – NVIDIA Corporation)



(reproduzido de "Chromatic Aberration"
Mark Kligard – CEDEC 2001)

Shaders

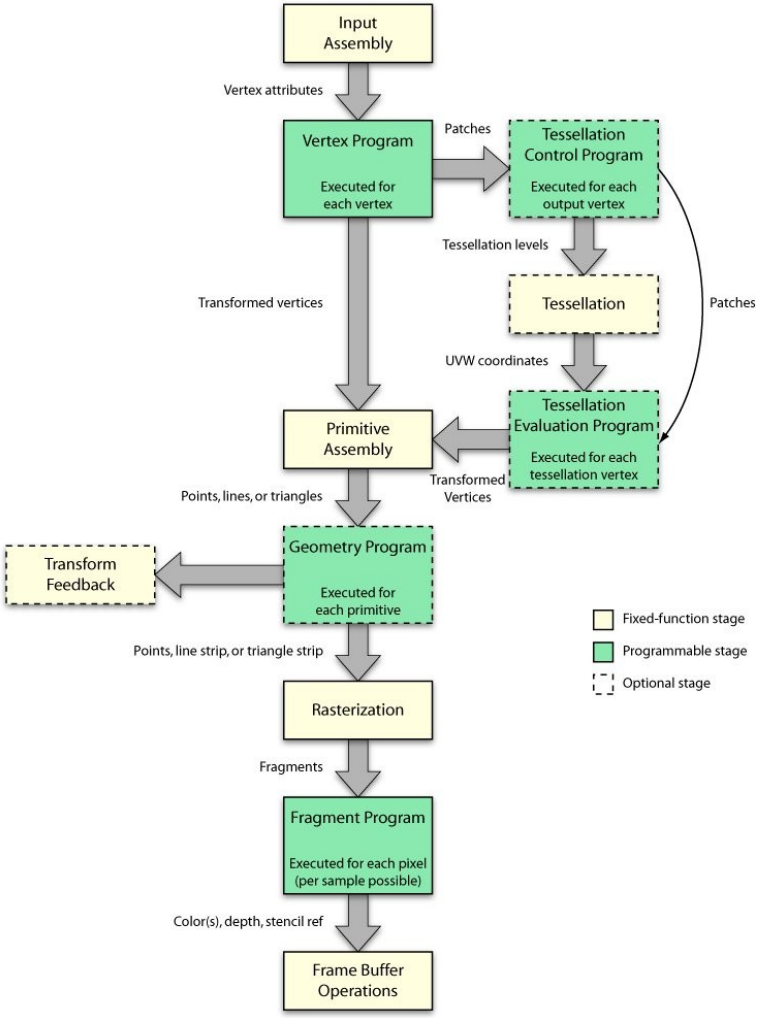


Pixar Toy Story - 1995

Shaders



GPU's



Pipeline Gráfico
(por Eric Lengyel - 2012)

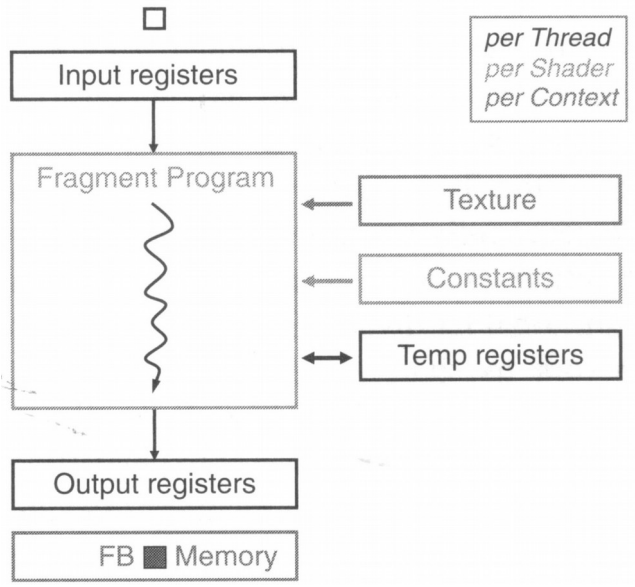
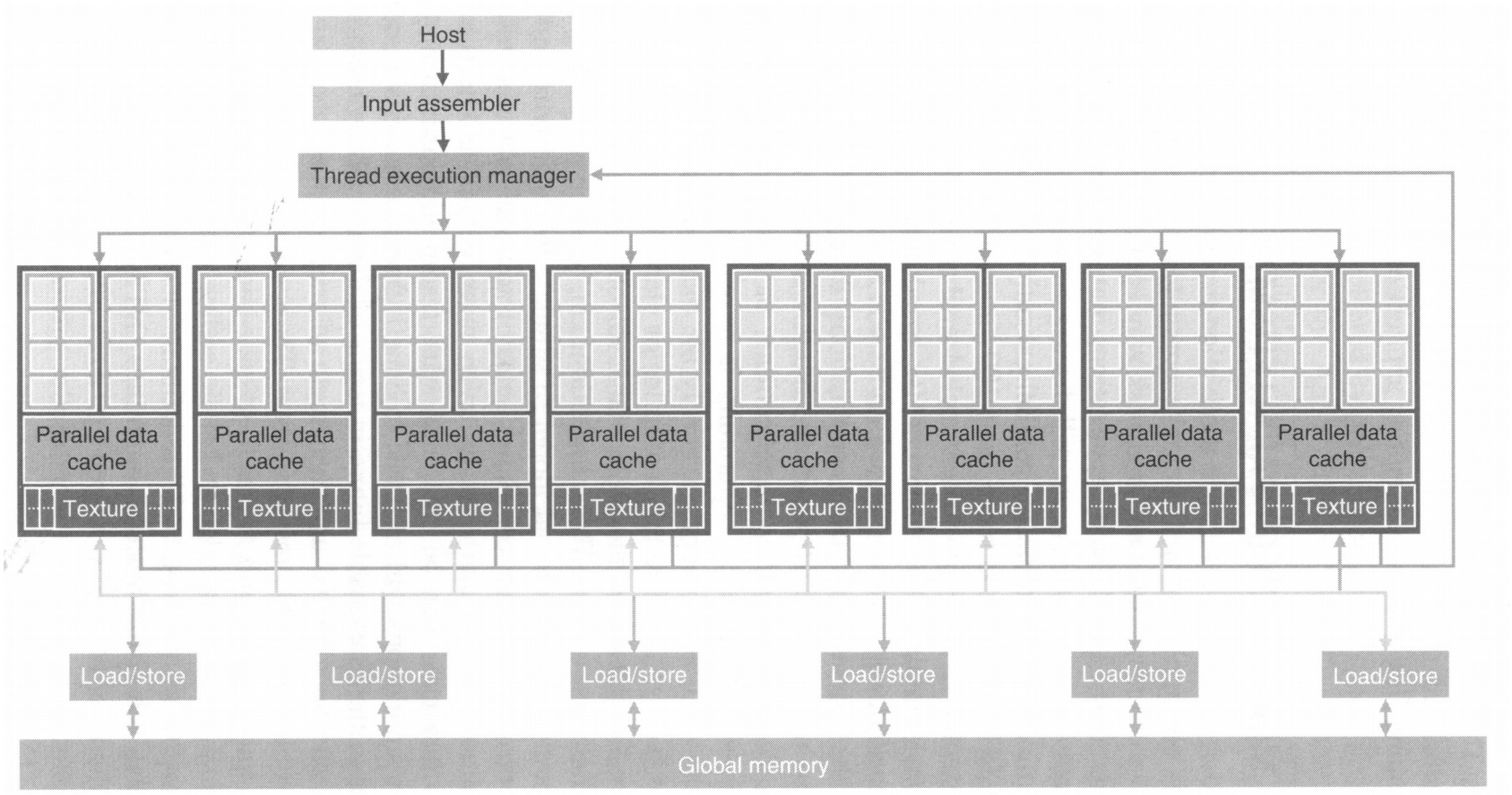


Diagrama de Shader NVidia
(de. “Programming Massively Parallel Processors”
Kirk, Hwu - NVIDIA)

GPU's



Arquitetura de GPU NVIDIA (2007)
(de. “Programming Massively Parallel Processors”
Kirk, Hwu - NVIDIA)

Computação Genérica em GPU's

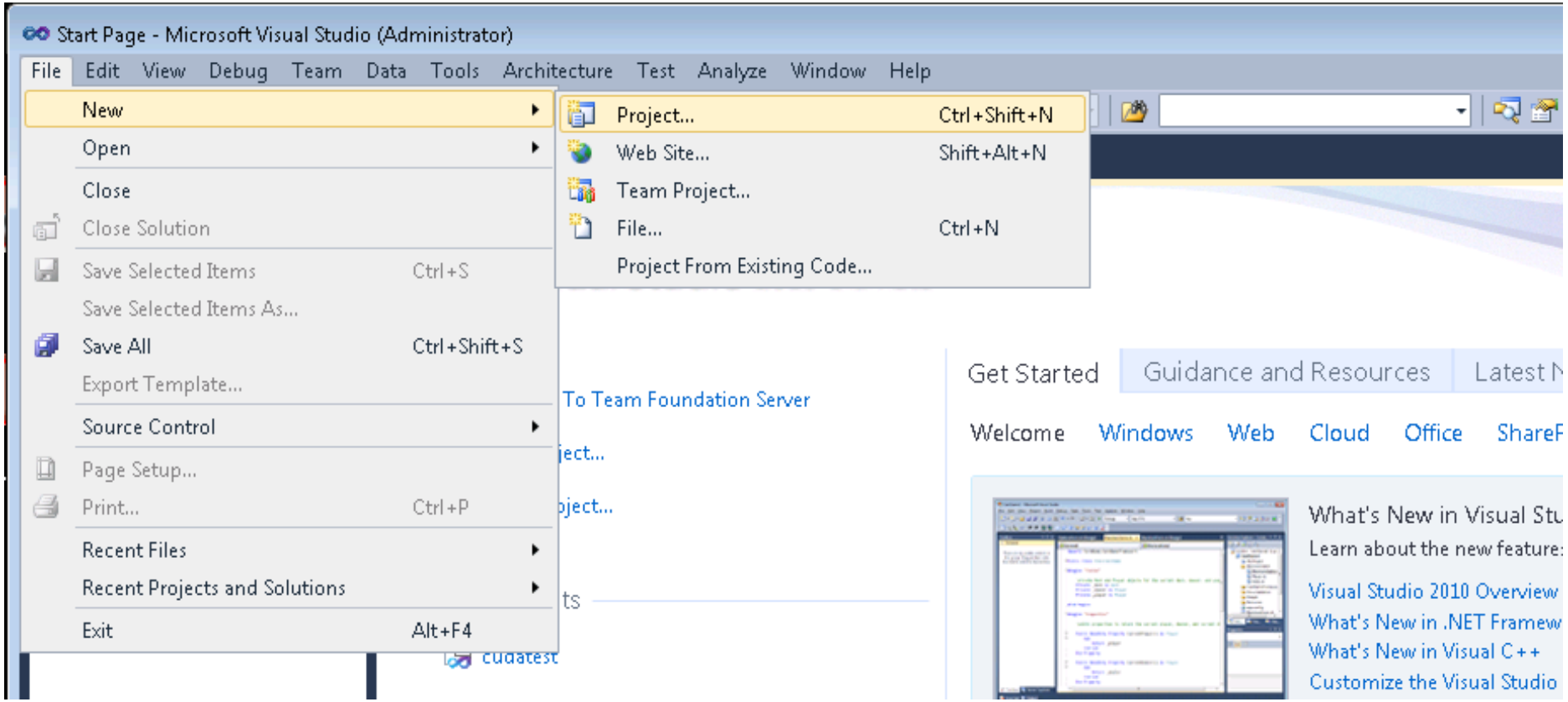
- CUDA (NVIDIA – 2007)
- OpenCL (Apple/Khronos 2008)
- DirectCompute (Microsoft - 2008)

Exercícios:

- Configurar Ambiente CUDA
- Recuperar Capacidades do Dispositivo
- Multiplicação de dois vetores
- Testes de desempenho

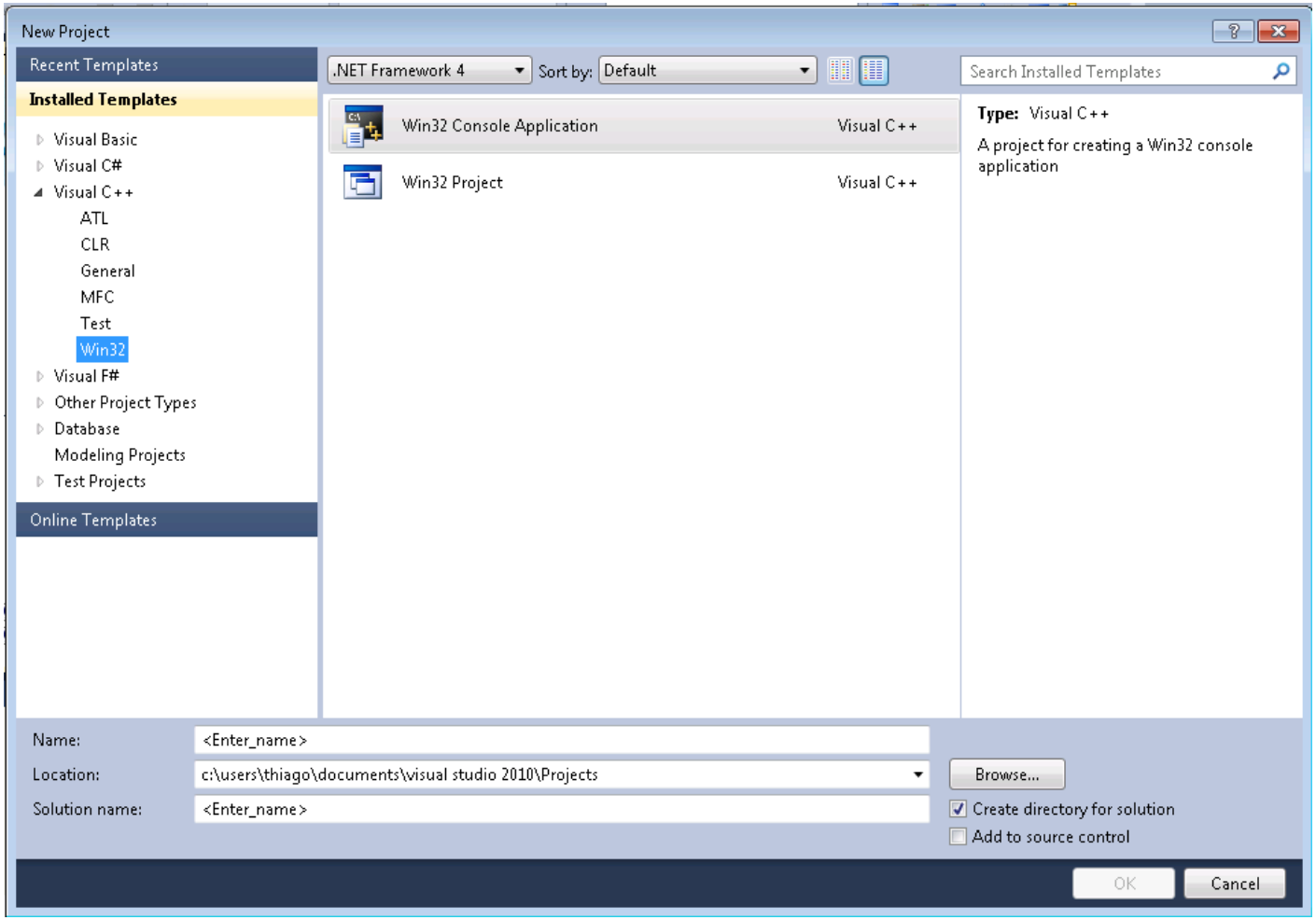
Configurar Ambiente CUDA

- Visual Studio 2010



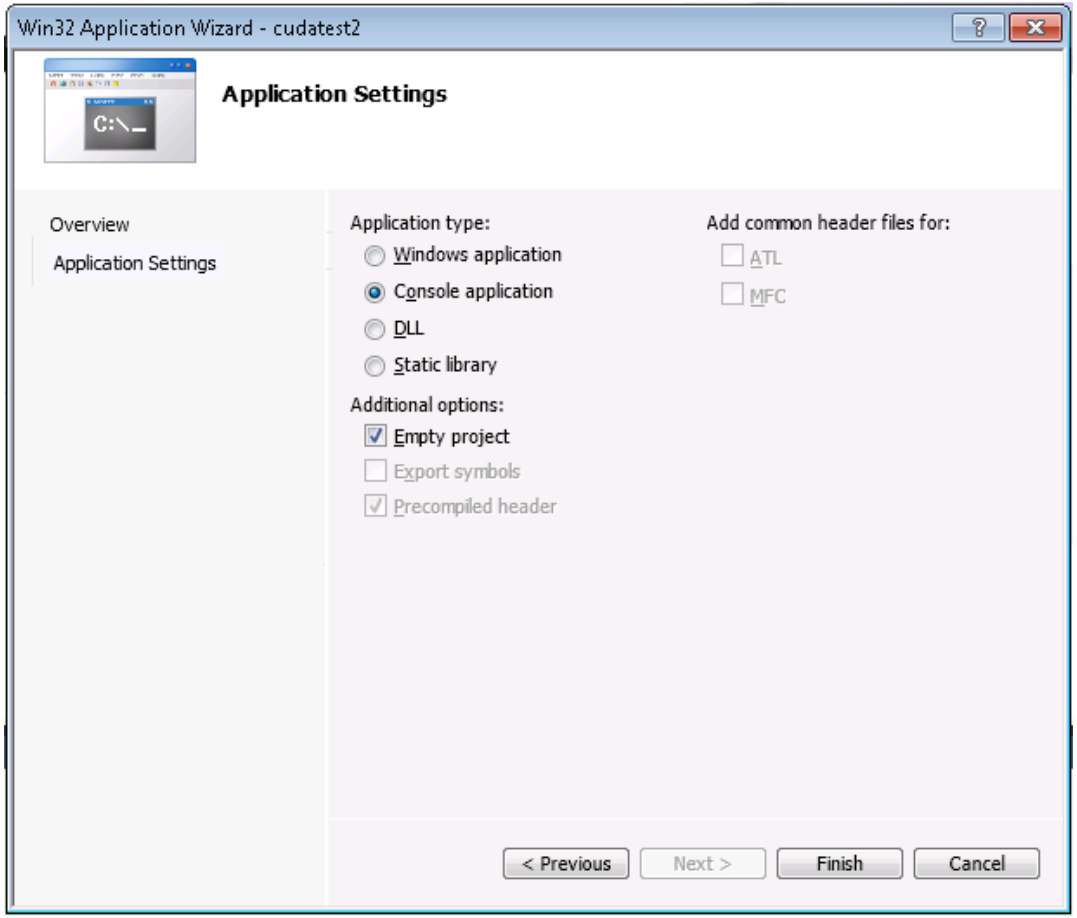
Configurar Ambiente CUDA

- Visual Studio 2010



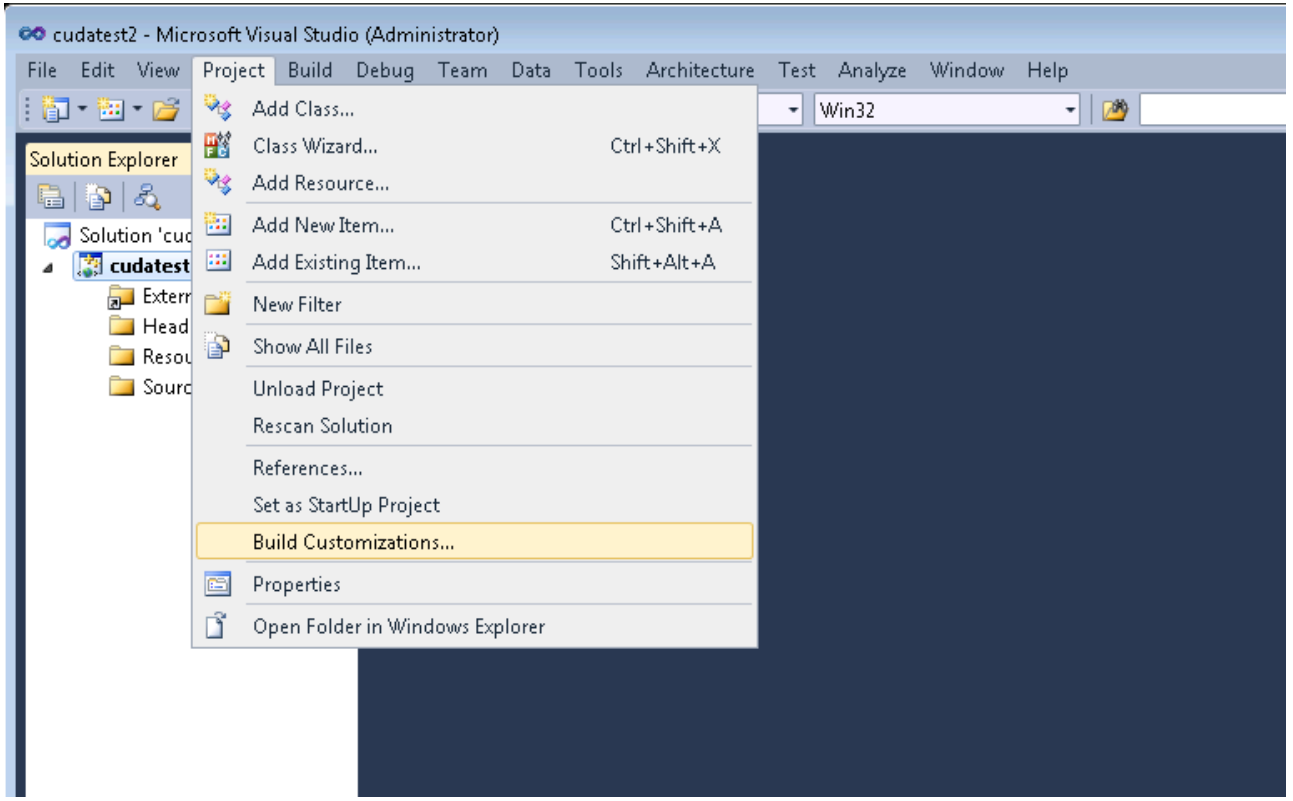
Configurar Ambiente CUDA

- Visual Studio 2010



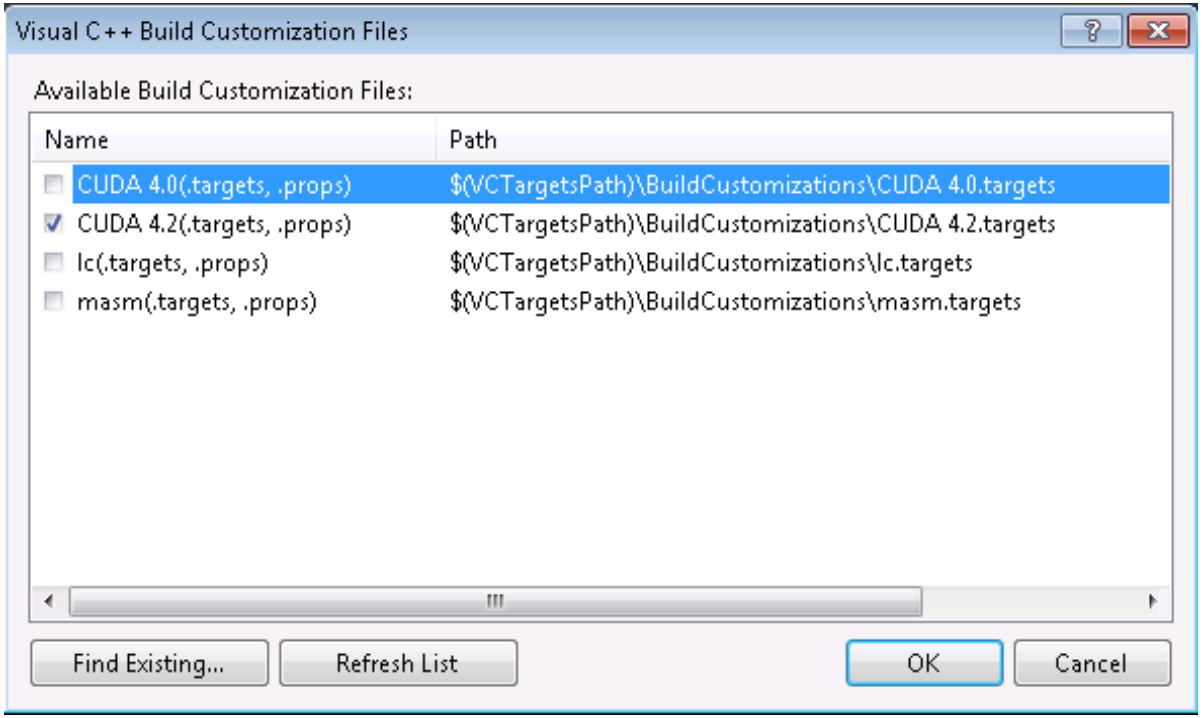
Configurar Ambiente CUDA

- Visual Studio 2010



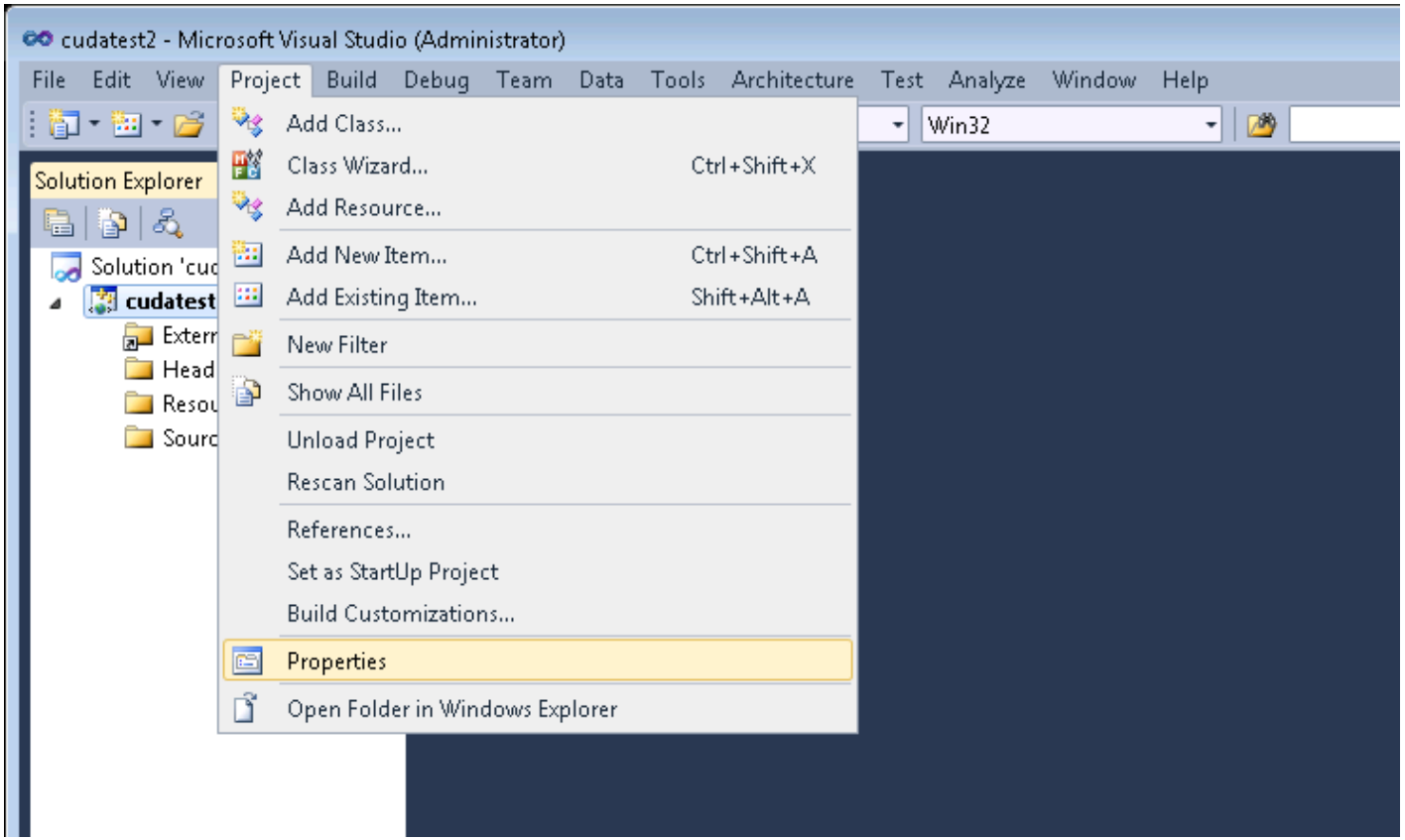
Configurar Ambiente CUDA

- Visual Studio 2010



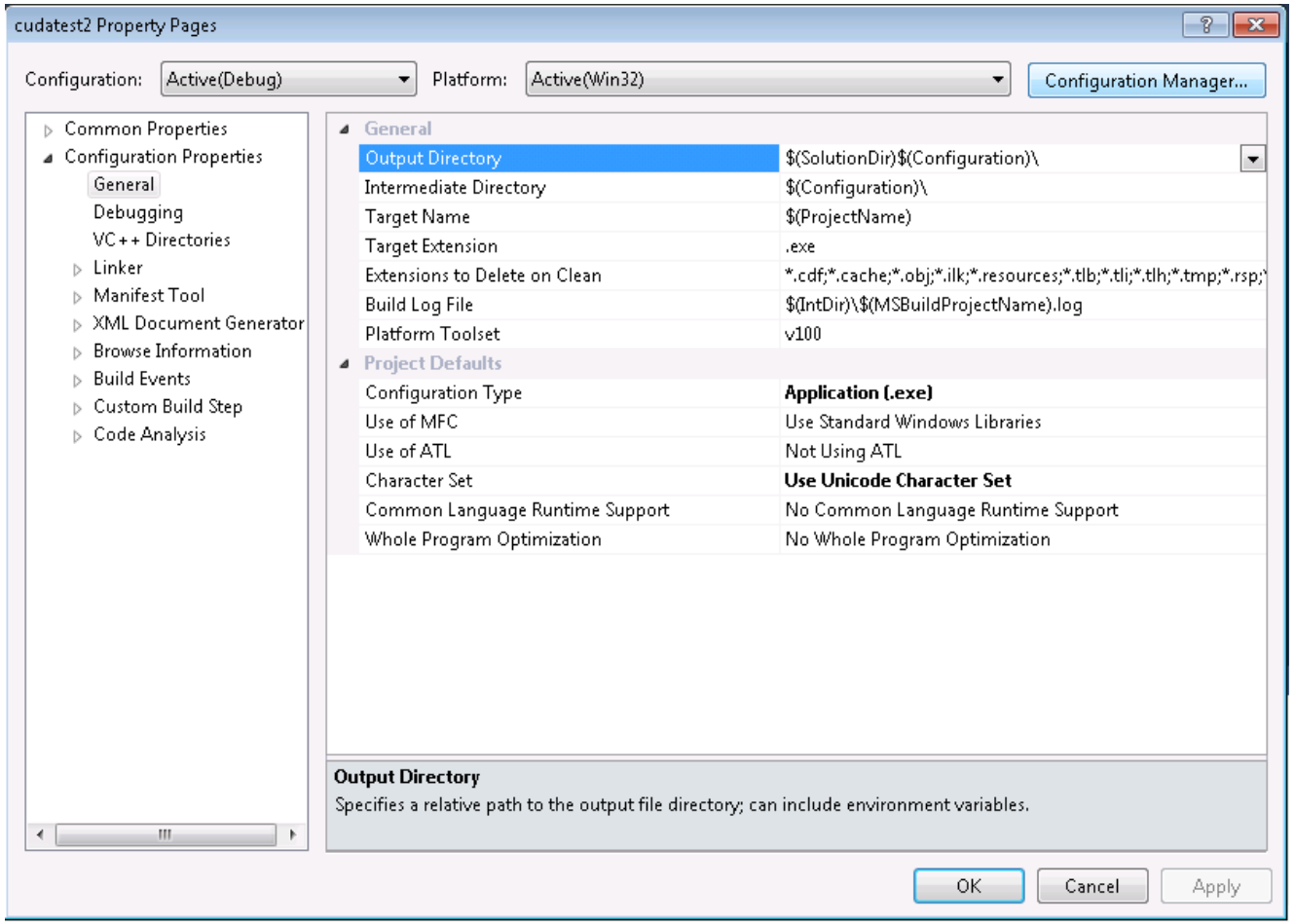
Configurar Ambiente CUDA

- Visual Studio 2010



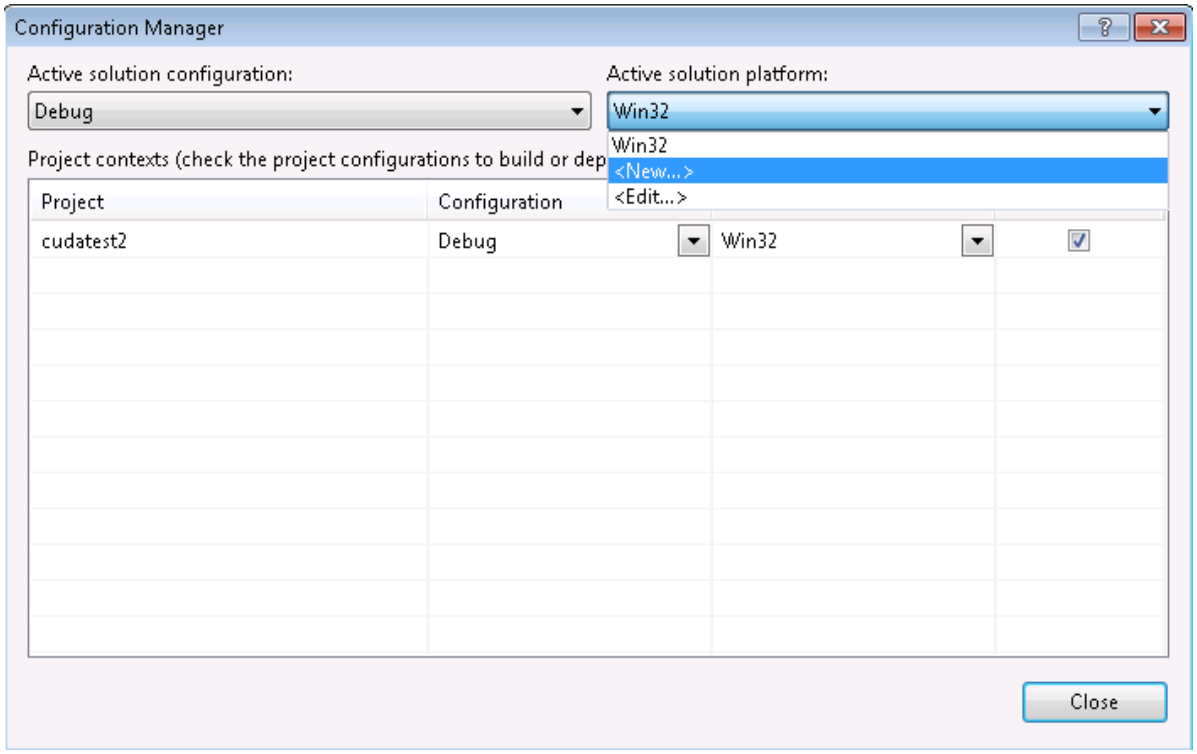
Configurar Ambiente CUDA

- Visual Studio 2010



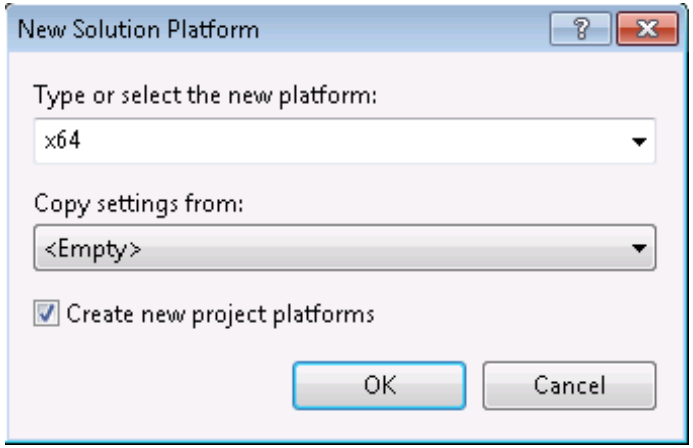
Configurar Ambiente CUDA

- Visual Studio 2010



Configurar Ambiente CUDA

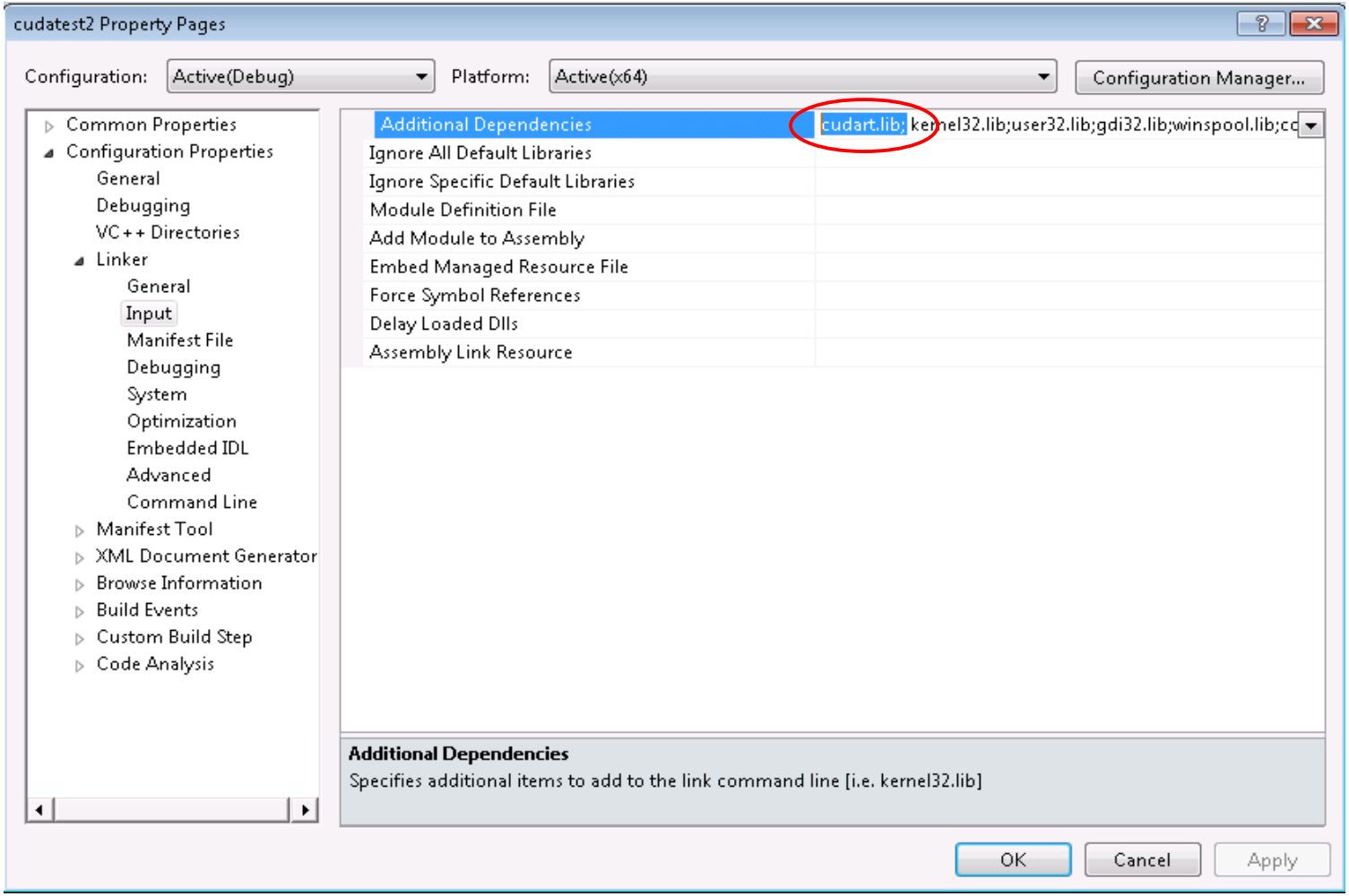
- Visual Studio 2010



Configurar Ambiente CUDA

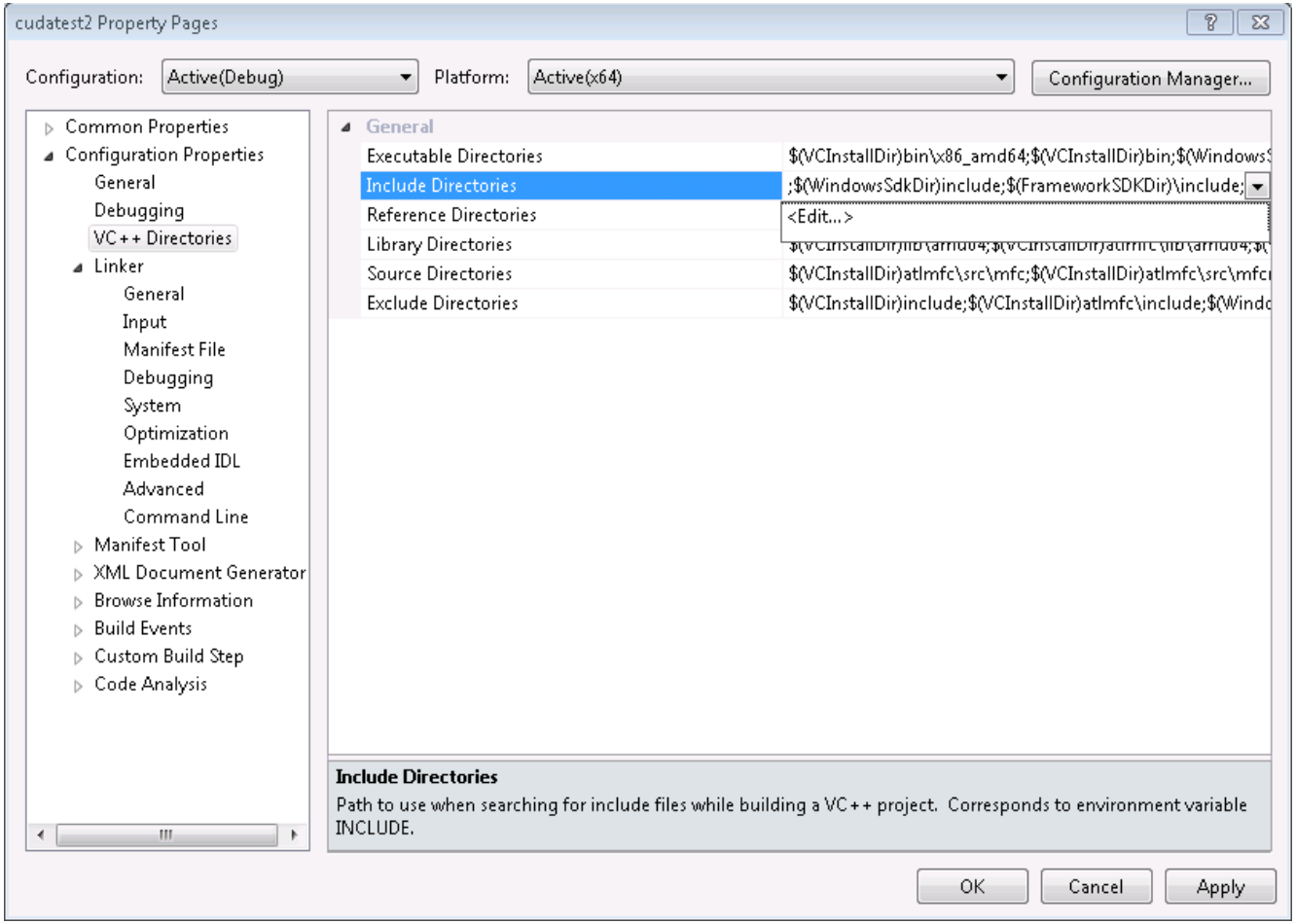
- Visual Studio 2010

cudart.lib



Configurar Ambiente CUDA

- Visual Studio 2010



Configurar Ambiente CUDA

- Recuperar Capacidades do Dispositivo

devicequery.cu

Exercícios

```
#include <iostream>
#include <nvidia_book/common/book.h>

__device__ int soma(int a, int b) { return a+b;}

__global__ void kernel(int a, int b, int *res)
{
    *res = soma(a, b);
}


int main(int argc, char **argv) {
    int res;
    int *device_res;

    HANDLE_ERROR( cudaMalloc( (void**)&device_res, sizeof(int)));

    kernel<<<1,1>>>(2,2,device_res);
    HANDLE_ERROR( cudaMemcpy( res, device_res, sizeof(int),
                              cudaMemcpyDeviceToHost ));

    std::cout << "2+2=" << res << std::endl;
    return 0;
}
```

Exercícios

```
__global__ void kernel(float *a, float *b, float *cresult)
{
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
```

```
kernel<<<blocks,threads>>>(da,db,dc);
```

Exercícios

```
int main(int argc, char **argv) {
    float a[N], b[N], c[N];
    float *da, *db, *dc;


    HANDLE_ERROR( cudaMalloc( (void**)&da, sizeof(float)*N ) );
    HANDLE_ERROR( cudaMalloc( (void**)&db, sizeof(float)*N ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dc, sizeof(float)*N ) );
```

```
    HANDLE_ERROR( cudaMemcpy( da, a, sizeof(float)*N,
                              cudaMemcpyHostToDevice ));
    HANDLE_ERROR( cudaMemcpy( db, b, sizeof(float)*N,
                              cudaMemcpyHostToDevice ));
```

```
    HANDLE_ERROR( cudaMemcpy( c, dc, sizeof(int)*N,
                              cudaMemcpyDeviceToHost ));
```

Multiplicação de vetores

$c_i = a_i b_i$

- Dimensão do vetor é uma macro global: #define N 1000
- Cada thread se ocupa de um único índice (por enquanto...)
- Derive i das coordenadas da grade
- Verifique limites...

Multiplicação de vetores

$c_i = a_i b_i$

- Dimensão do vetor é uma macro global: #define N 1000
- Cada thread se ocupa de um único índice (por enquanto...)
- Derive i das coordenadas da grade
- Verifique limites...

  - Você é capaz de tratar de múltiplas posições por thread? Isso é útil?

Multiplicação de vetores

- Medir desempenho

```
cudaEvent_t start, stop;
HANDLE_ERROR( cudaEventCreate(&start));
HANDLE_ERROR( cudaEventCreate(&stop));
```

```
HANDLE_ERROR(cudaEventRecord(start,0));
```

→ Chamada do *kernel*

```
HANDLE_ERROR(cudaEventRecord(stop,0));
HANDLE_ERROR(cudaEventSynchronize(stop));
```

```
float time;
HANDLE_ERROR(cudaEventElapsedTime(&time, start, stop));

std::cout << "tempo:" << time << std::endl;
```

Multiplicação de vetores

- Medir desempenho
- Verifique com N grande (>30000)
- Compare com tempo de CPU

```
#include “gettime.h”
```

```
time = get_time();
for(int i=0;i<N;i++)
  c[i] = a[i]*b[i];
time = get_time() - time;

std::cout << "time:" << time << std::endl;
```

Multiplicação de vetores

- Medir desempenho
- Verifique com N grande (>30000)
- Compare com tempo de CPU

Mas...

Se incluirmos o tempo de cópia para a memória da GPU?

Multiplicação de vetores

- Medir desempenho
- Verifique com N grande (>30000)
- Compare com tempo de CPU

Mas...

Se incluirmos o tempo de cópia para a memória da GPU?

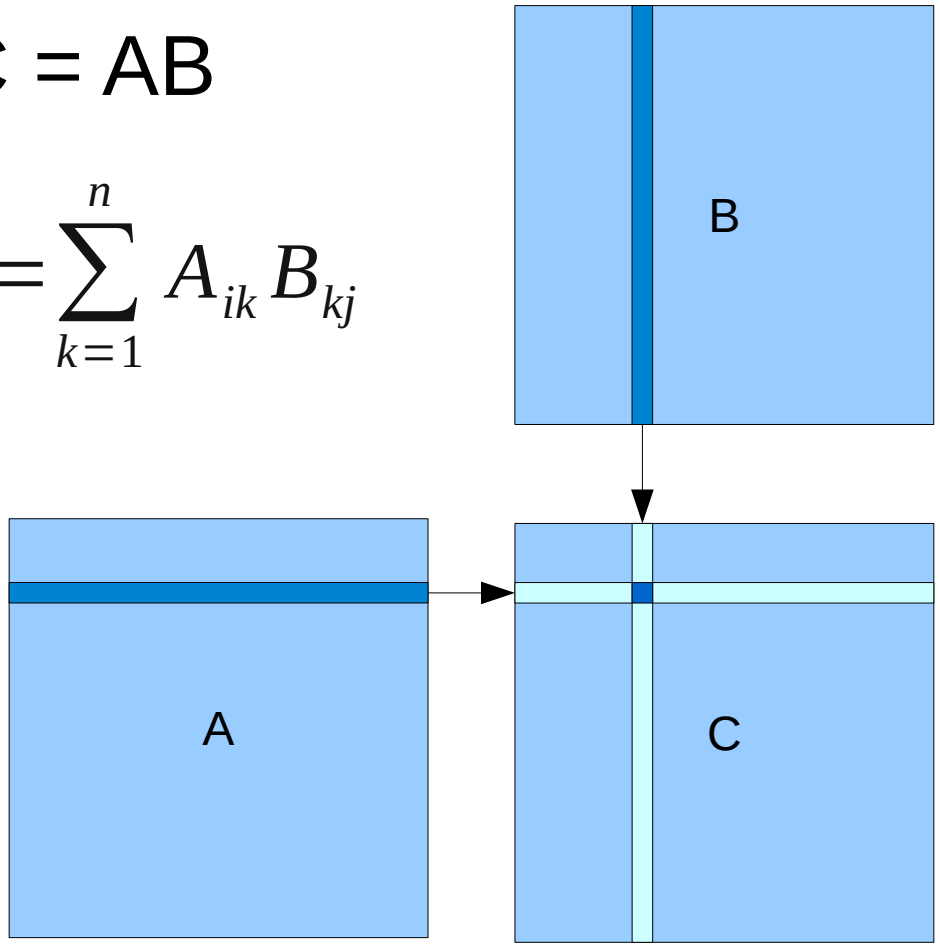
O(n) movimentos para memória da GPU

O(n) operações com ponto flutuante

Multiplicação de matrizes

C = AB

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$



Multiplicação de matrizes

C = AB

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$O(n^2)$ movimentos para memória da GPU

$O(n^3)$ operações com ponto flutuante

Multiplicação de matrizes

Linearização de espaço

i+Nj

4x4:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

```
#define INDEX(i,j) ((i)+(j)*N)
```

Multiplicação de matrizes

1. Implementar algoritmo serializado (CPU)

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

```
  for(int i=0;i<N;i++)
     for(int j=0;j<N;j++) {
```

```
      }
```

Multiplicação de matrizes

2. Implementar algoritmo paralelo (GPU)

32*32=1024 threads por bloco

```
   dim3 blockDim(32,32);
   dim3 gridDim((N+31)/32,(N+31)/32);
```

Uma thread por posição do resultado:

```
__global__ void mul(float *a, float *b, float *c)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x;
    int j = threadIdx.y + blockIdx.y*blockDim.y;
```

Multiplicação de matrizes

- Medir desempenho
- Compare com tempo de CPU, *INCLUINDO* o tempo de cópia para a memória da GPU!
- E agora?

Redução

- $f(\{a_1, a_2, a_3, \ldots, a_n\}) \rightarrow X$
- Associatividade:

  $f(\{a_1, a_2, a_3, \ldots, a_n\}) =$

  $f(f\{a_1, \ldots, a_j\}), f\{a_{j+1}, \ldots, a_n\}) =$

  - $f(f\{a_1, \ldots, a_k\}), f\{a_k, \ldots, a_n\})$

Redução

- f({a_1, a_2, a_3, …, a_n}) → X
- Associatividade:

  f({a_1, a_2, a_3, …, a_n})=

  f(f{a_1, …, a_j}),f{a_{j+1},…, a_n})=

  - f(f{a_1, …, a_k}),f{a_k,…, a_n})

- Exemplo: soma

$$f\left(a_{1,}\ldots,a_n\right)=\sum_{i=1}^{n} a_i$$

Redução: Soma

- Sequencial



1. Res ← 0
2. Res += a_1
3. Res += a_2
4. ...

O(n) operações com ponto flutuante

Redução: Soma

- Paralela?



Redução: Soma

- Paralela!

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
a_1+a_2		a_3+a_4		a_5+a_6		a_7+a_8	
$a_1+a_2+a_3+a_4$				$a_5+a_6+a_7+a_8$			
$a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8$							

Redução: Soma

- Paralela!

a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8
a_1+a_2		a_3+a_4		a_5+a_6		a_7+a_8	
$a_1+a_2+a_3+a_4$				$a_5+a_6+a_7+a_8$			
$a_1+a_2+a_3+a_4+a_5+a_6+a_7+a_8$							

O(log(n)) operações com ponto flutuante!

Redução: Soma

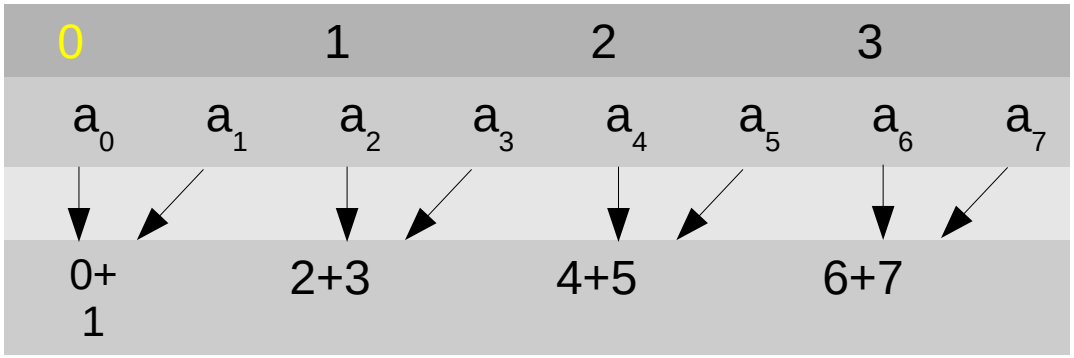
- Algumas simplificações:
  - Tamanho do vetor é potência de 2
  - Apenas UM bloco
  - N/2 threads
  - Usar memória local (__shared__)

Redução: Soma

- Kernel:
  - Copiar buffer para memória local
    - Como temos N/2 threads, cada thread copia duas posições
    - Evitar sobreposição!

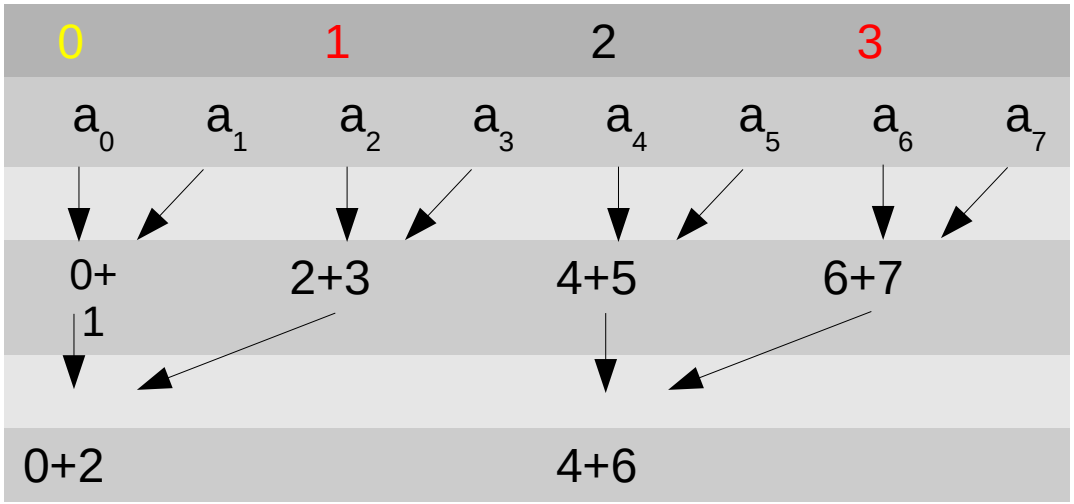
Redução: Soma

- Kernel:
  - Soma



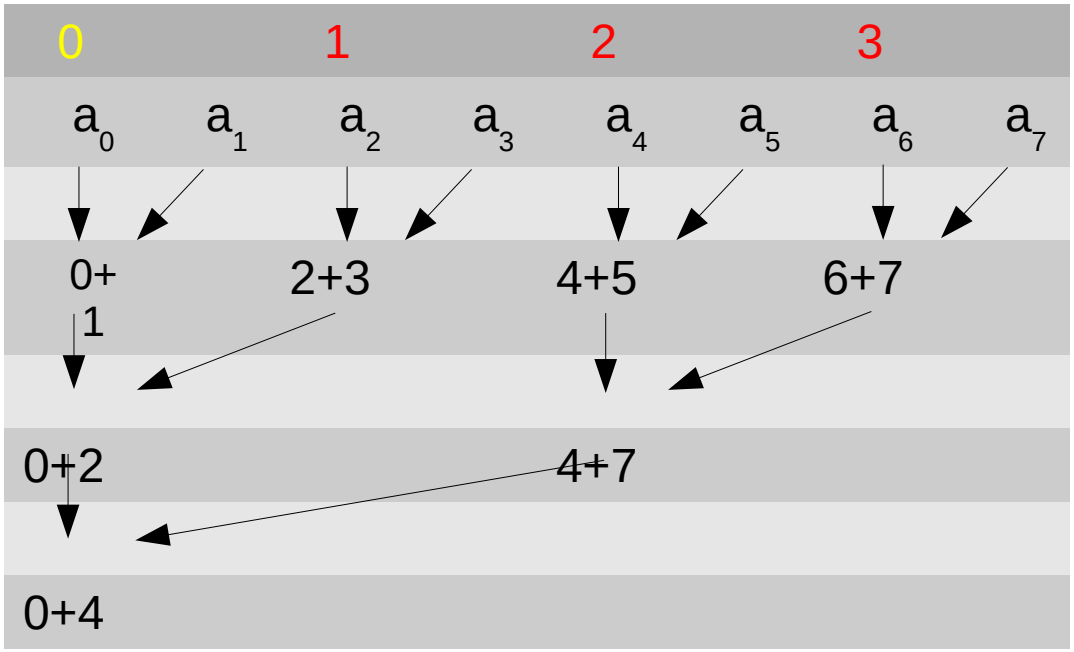
Redução: Soma

- Kernel:
  - Soma



Redução: Soma

- Kernel:
  - Soma



Redução: Soma

- Kernel:
  - Soma
    - Delta duplica a cada iteração

```
for(int delta = 1; delta <= N/2; delta*=2) {
```

    - Metade das threads *não* trabalha na iteração seguinte!

```
__syncthreads();
if((tid % delta)==0)
```

Redução: Soma

- Kernel:
  - Resultado
    - Apenas UMA thread copia o resultado de volta

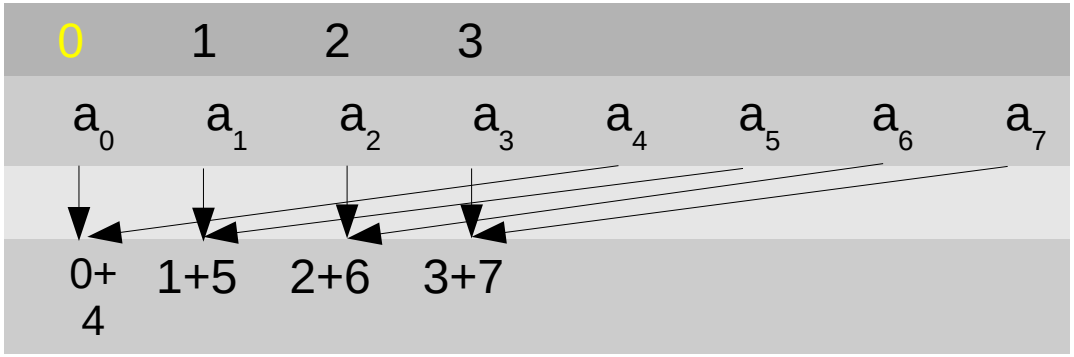
Redução: Soma

- Desempenho?
  - Divergência de threads dentro de um *warp*

0		1		2		3	
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
0+1		2+3		4+5		6+7	
0+2				4+7			

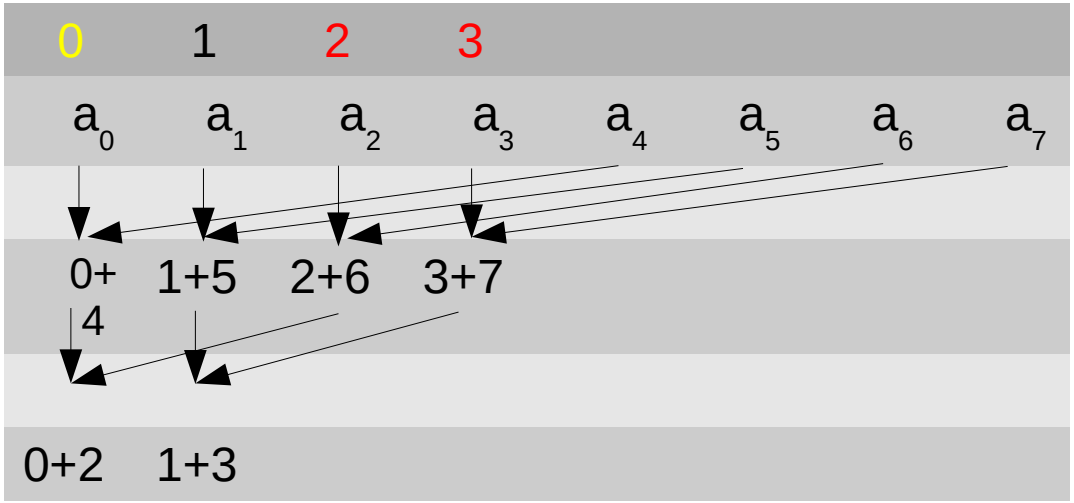
Redução: Soma

- Novo Kernel:



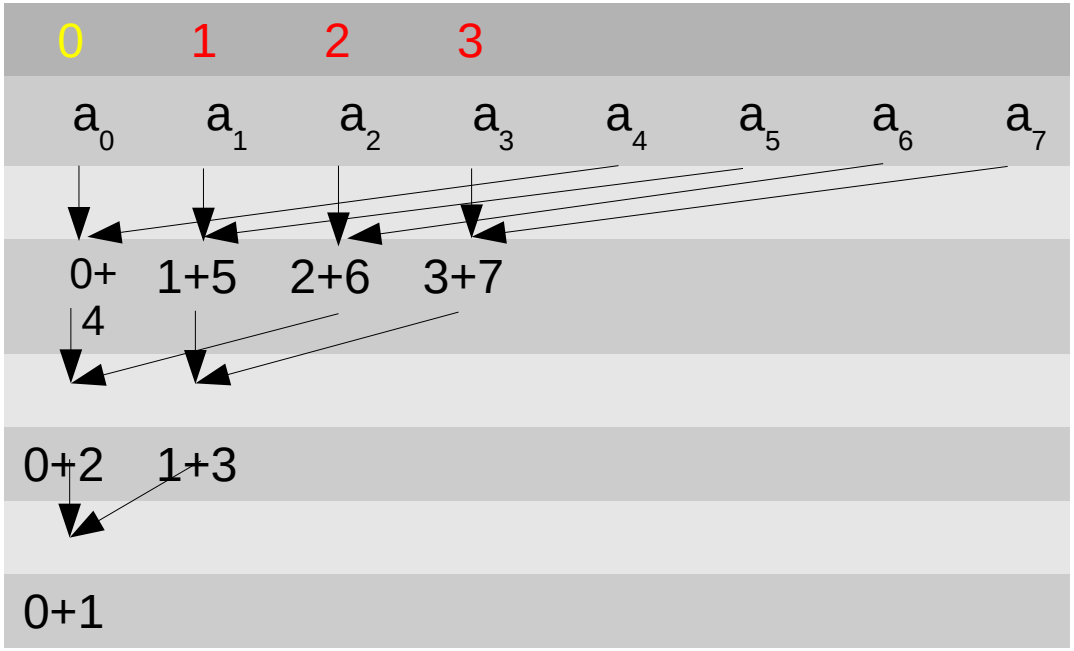
Redução: Soma

- Novo Kernel:



Redução: Soma

- Novo Kernel:



Redução: Soma

- Novo Kernel:
  - Delta cai pela metade a cada iteração

```
for(int delta = N/2; delta > 0; delta/=2) {
```

  - Metade superior das threads *não* trabalha na iteração seguinte!

```
__syncthreads();
if(tid<delta)
```

Redução: Soma

- Desempenho?
  - N ainda muito pequeno!
  - Como fazer para totalizar vetores maiores?
    - __syncthreads NÃO funciona entre blocos!
    - Podemos duplicar o número de elementos totalizando E inicializando shared

```
__shared__ float partial[N/2];

__global__ void sum(float *a, float *c)
{
    int tid = threadIdx.x;
    partial[tid] = a[tid]+a[tid+N/2];
    for(int delta = N/4; delta > 0; delta/=2) {
```

Redução: Soma

- Desempenho?
  - Podemos duplicar o número de elementos totalizando E inicializando shared

```
#define N 1024

__shared__ float partial[N/2];

__global__ void sum(float *a, float *c)
{
    int tid = threadIdx.x;
    partial[tid] = a[tid]+a[tid+N/2];
    for(int delta = N/4; delta > 0; delta/=2) {
```

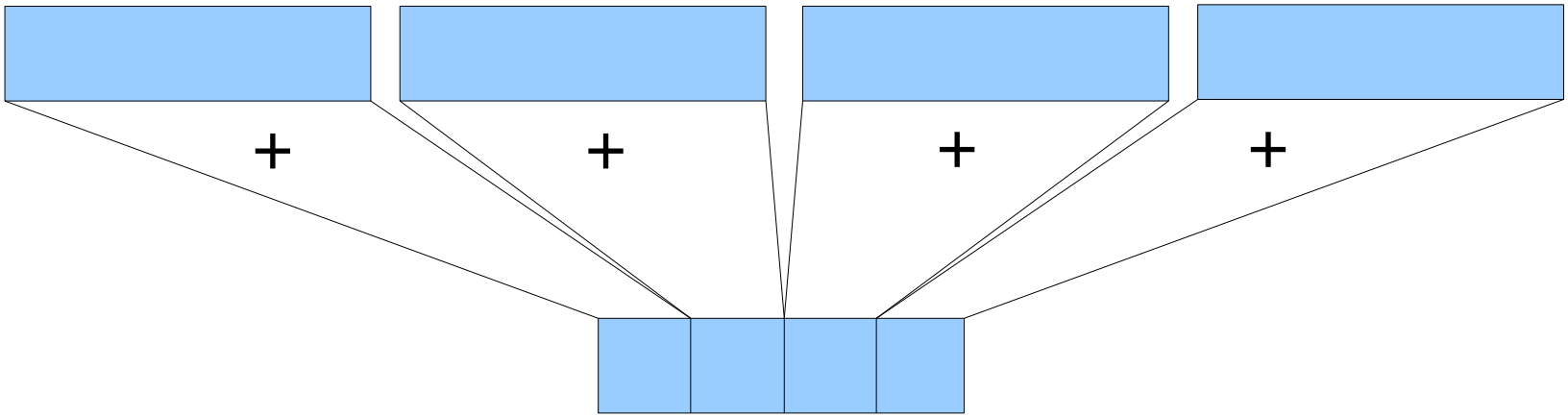
  - Mas o ganho é pequeno ainda...
  - Na verdade, N~1024 ainda é muito pouco para a GPU!
  - Como fazer para totalizar vetores maiores?

Redução: Soma

- Como fazer para totalizar vetores ainda maiores?

Redução: Soma

- Como fazer para totalizar vetores ainda maiores?
  - Dividir-para-conquistar



Redução: Soma

- Vetores grandes
- 2 kernels

```
__global__ void partialsum(float *a, float *subtotals)
```

```
__global__ void totalSum(float *subtotals, float *c)
```

Redução: Soma

- Vetores grandes
- PartialSum
  - Selecionar a região do buffer de entrada a ser totalizada



Redução: Soma

- Vetores grandes
- PartialSum
  - Selecionar a região do buffer de entrada a ser totalizada

```
__global__ void partialsum(float *a, float *subtotals)
{
    int tid = threadIdx.x;
    a = a + 2*NTHREADS*blockIdx.x;
```



Redução: Soma

- Vetores grandes
- PartialSum
  - Escrever a soma parcial na posição adequada de subtotals

Redução: Soma

- Vetores grandes
- PartialSum
  - Escrever a soma parcial na posição adequada de subtotals

```
    if(threadIdx.x==0)
      subtotals[blockIdx.x] = partial[0];
}
```

Redução: Soma

- Vetores grandes
  - totalSum
    - Ler e totalizar 2-a-2 os valores em subtotals e escrevê-los em partial

Redução: Soma

- Vetores grandes
  - totalSum
    - Ler e totalizar 2-a-2 os valores em subtotals e escrevê-los em partial

```
__global__ void totalSum(float *subtotals, float *c)
{
    int tid = threadIdx.x;
    partial[tid] = subtotals[tid];
    partial[tid] += subtotals[tid+SUBTOTAL_SIZE/2];
```

Redução: Soma

- Vetores grandes
  - totalSum
    - Efetuar as somas parciais

Redução: Soma

- Vetores grandes
  - totalSum
    - Efetuar as somas parciais

```
for(int delta = SUBTOTAL_SIZE/4; delta > 0; delta/=2) {
     __syncthreads();
     if(tid<delta)
       partial[tid] += partial[tid+delta];
    }
```

Redução: Soma

- Vetores grandes
  - totalSum
    - Efetuar as somas parciais

```
for(int delta = SUBTOTAL_SIZE/4; delta > 0; delta/=2) {
     __syncthreads();
     if(tid<delta)
       partial[tid] += partial[tid+delta];
    }
```

Redução: Soma

- Vetores grandes
  - totalSum
    - Escrever o resultado final em c

Redução: Soma

- Vetores grandes
  - totalSum
    - Escrever o resultado final em c

```
    if(threadIdx.x==0)
      *c = partial[0];
}
```

Redução: Soma

- Desempenho?

Redução: Encontrar máximo

- $f(\{a_1, a_2, a_3, \ldots, a_n\}) \rightarrow X$
- Associatividade:

  $f(\{a_1, a_2, a_3, \ldots, a_n\}) =$

  $f(f\{a_1, \ldots, a_j\}), f\{a_{j+1}, \ldots, a_n\}) =$

  - $f(f\{a_1, \ldots, a_k\}), f\{a_k, \ldots, a_n\})$

Redução: Encontrar máximo

- Associatividade:

```
class pair {
  int pos;
  float val;
public:
  __device__ pair(int pos, float val): pos(pos), val(val){}
  __device__ pair(){};
  __device__ int indice() const { return this->pos; }

  __device__ static pair getLarger(pair a, pair b) {
```

```
  }
};
```

Redução: Encontrar máximo

- Associatividade:

```
class pair {
  int pos;
  float val;
public:
  __device__ pair(int pos, float val): pos(pos), val(val){}
  __device__ pair(){};
  __device__ int indice() const { return this->pos; }

  __device__ static pair getLarger(pair a, pair b) {
```

```
  }
};
```

Redução: Encontrar máximo

- Associatividade:

```
class pair {
  int pos;
  float val;
public:
  __device__ pair(int pos, float val): pos(pos), val(val){}
  __device__ pair(){};
  __device__ int indice() const { return this->pos; }

  __device__ static pair getLarger(pair a, pair b) {
    if(a.val>b.val)
      return a;
    return b;
  }
};
```

Redução: Encontrar máximo

- Inicializar partial:
  - Criar objetos pair e reduzir 2-a-2 em partial

Redução: Encontrar máximo

- Inicializar partial:
  - Criar objetos pair e reduzir 2-a-2 em partial

```
partial[tid] = pair::getLarger(
            pair(tid,a[tid]),
            pair(tid+N/2,a[tid+N/2]));
```

Redução: Encontrar máximo

- Passo de redução: invocar getLarger!

Redução: Encontrar máximo

- Passo de redução: invocar getLarger!

```
partial[tid] = pair::getLarger(partial[tid],partial[tid+delta]);
```

Redução: Encontrar máximo

- Recuperar o resultado

Redução: Encontrar máximo

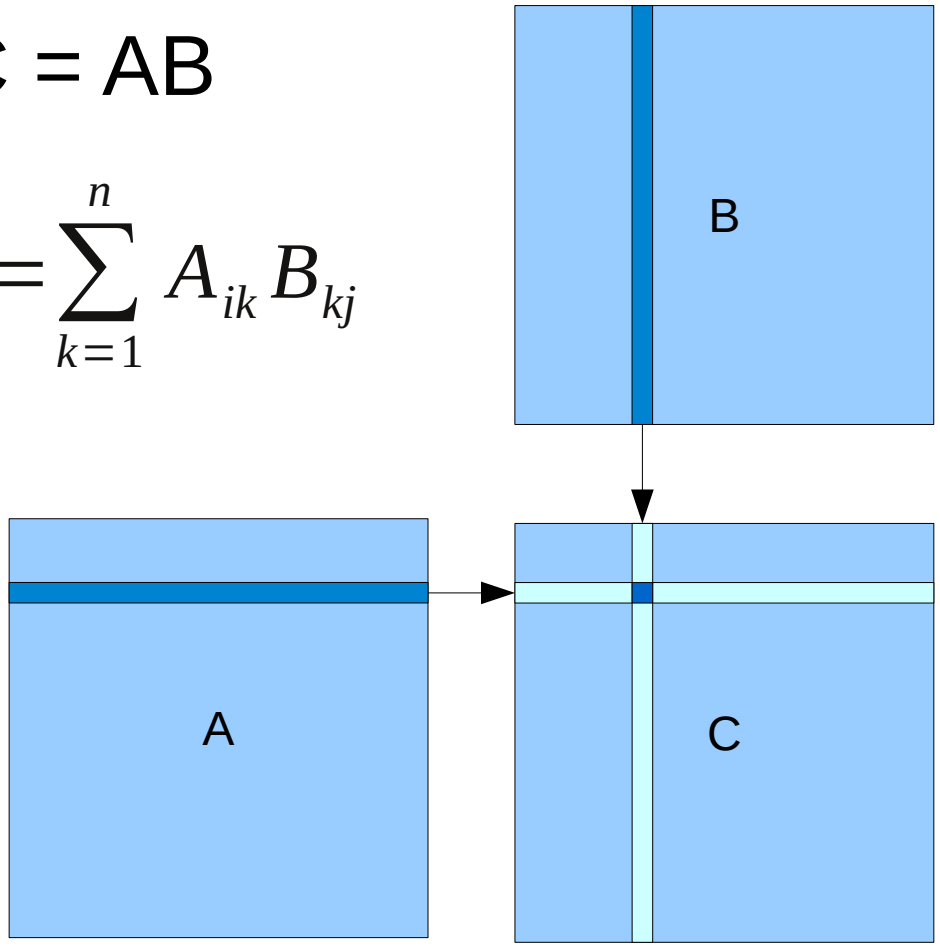
- Recuperar o resultado

```
*c = partial[0].indice();
```

Multiplicação de matrizes

C = AB

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$



Multiplicação de matrizes

C = AB

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$O(n^2)$ movimentos para memória da GPU

$O(n^3)$ operações com ponto flutuante

$O(n^3)$ operações de acesso à memória da GPU!

Multiplicação de matrizes

C = AB

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$O(n^2)$ movimentos para memória da GPU

$O(n^3)$ operações com ponto flutuante

$O(n^3)$ operações de acesso à memória da GPU!

Multiplicação de matrizes

Memória *Shared*

$O(n^3)$ operações de acesso à memória da GPU!

Multiplicação de matrizes

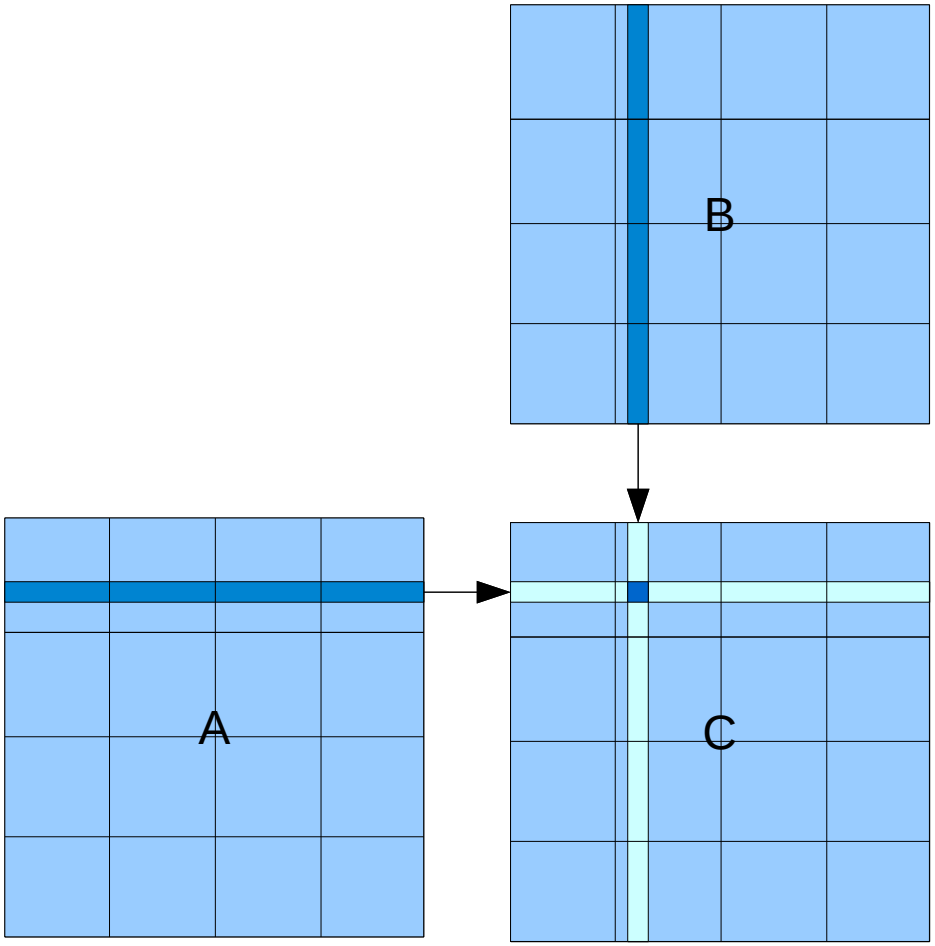
Memória *Shared*
→ Local ao bloco

$O(n^3)$ operações de acesso à memória da GPU, mas...

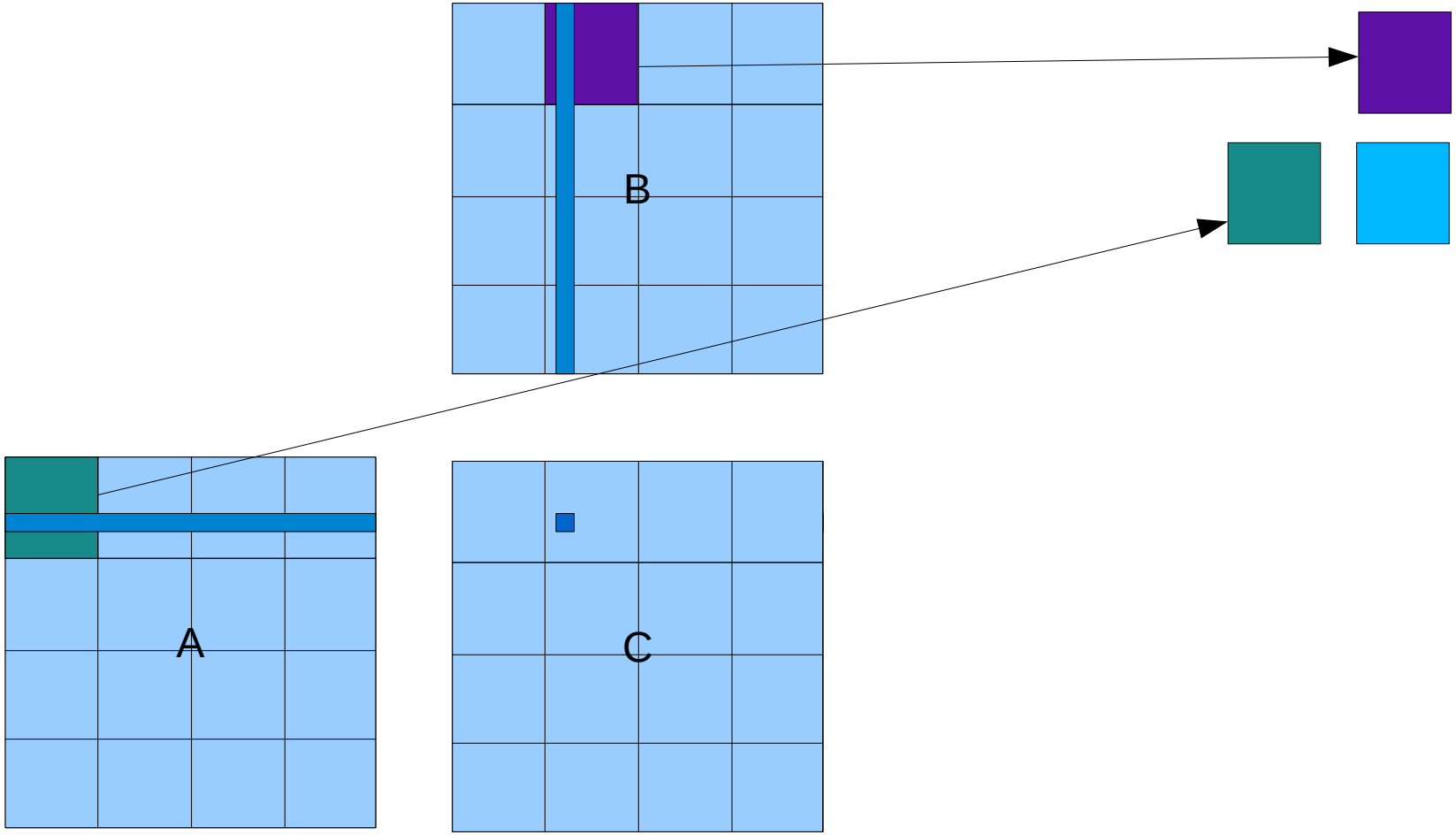
$O(n^2)$ operações de acesso à memória global (lenta)

$O(n^3)$ operações de acesso à memória *shared* (rápida)

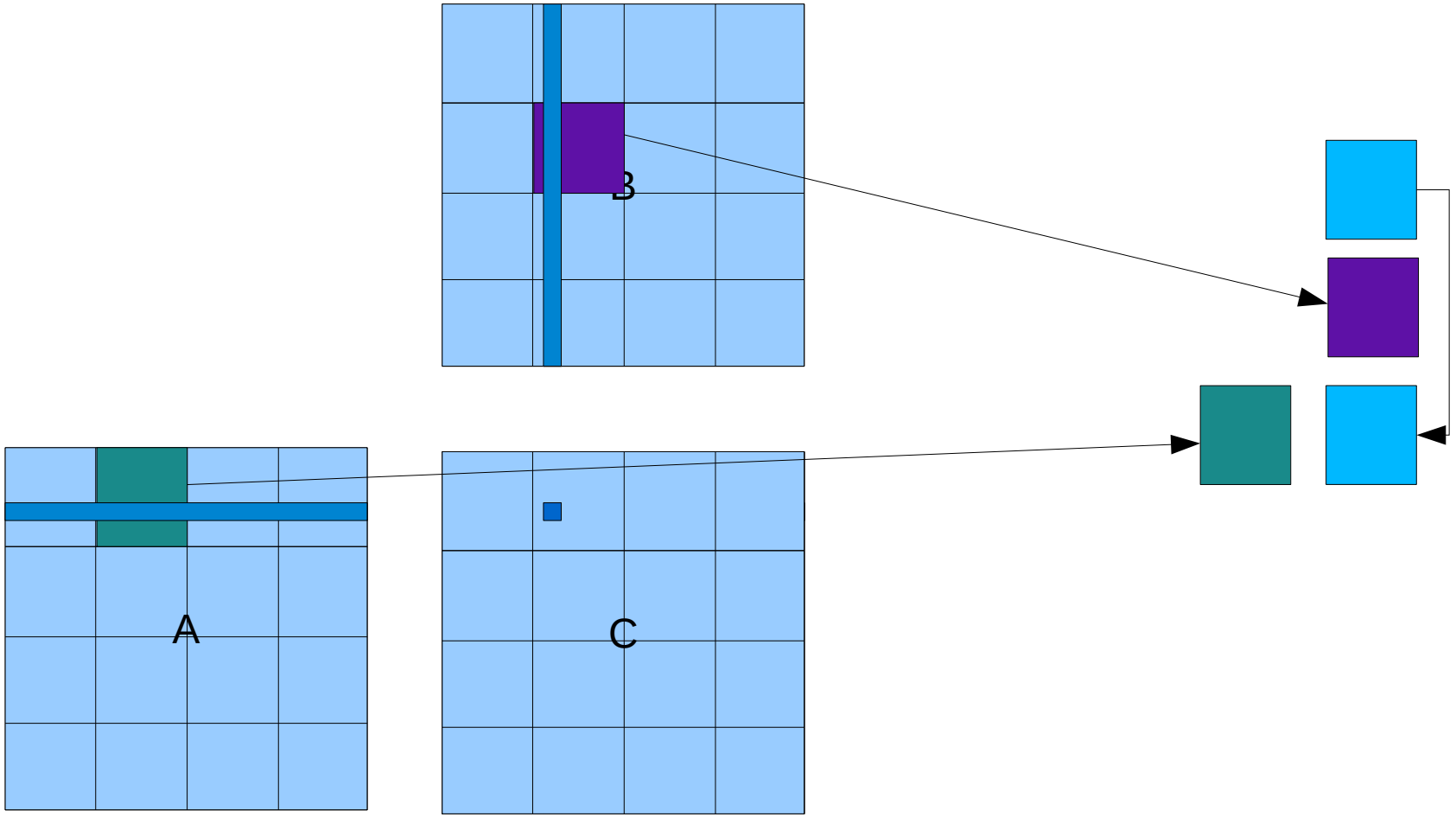
Multiplicação de matrizes



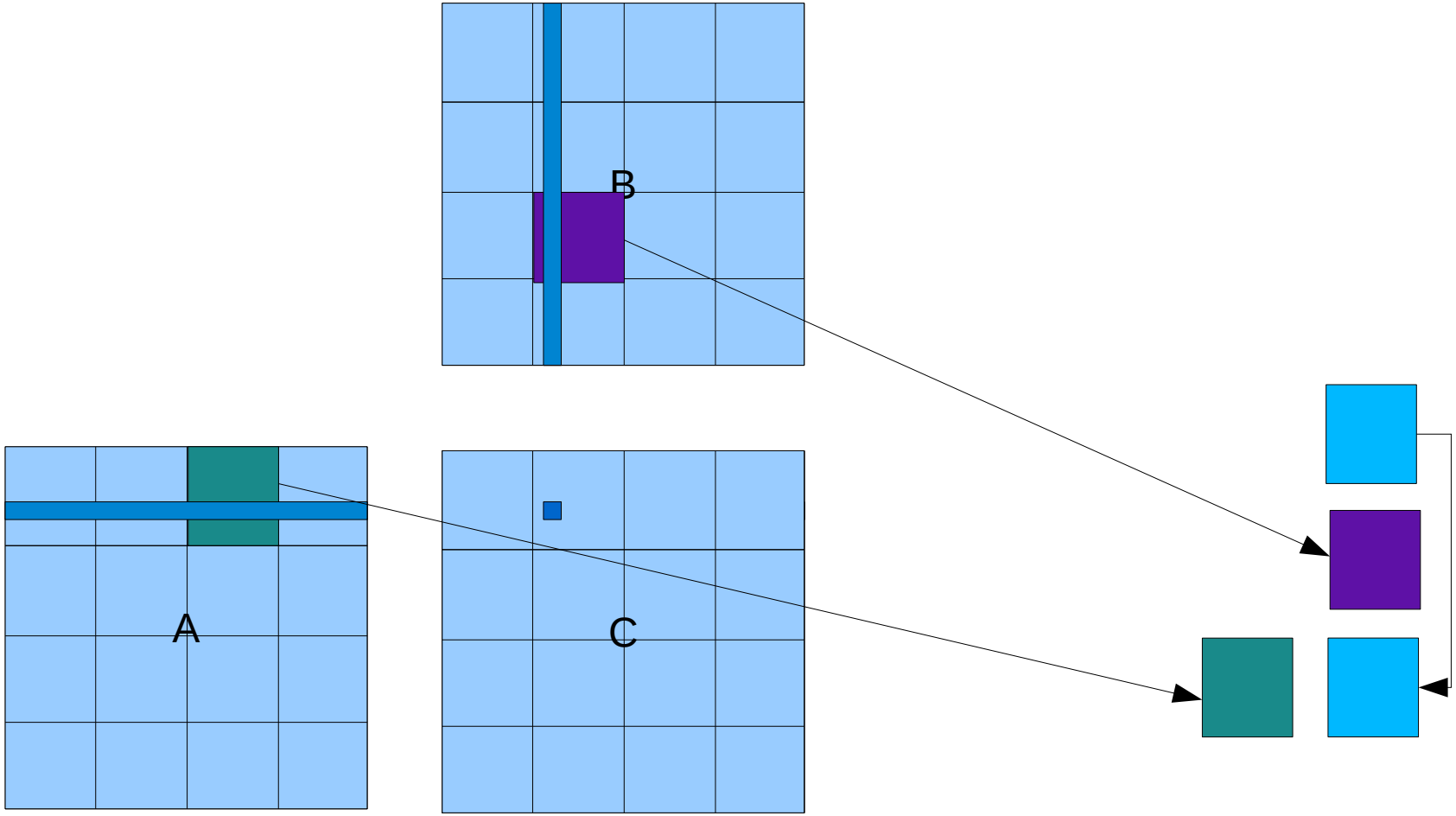
Multiplicação de matrizes



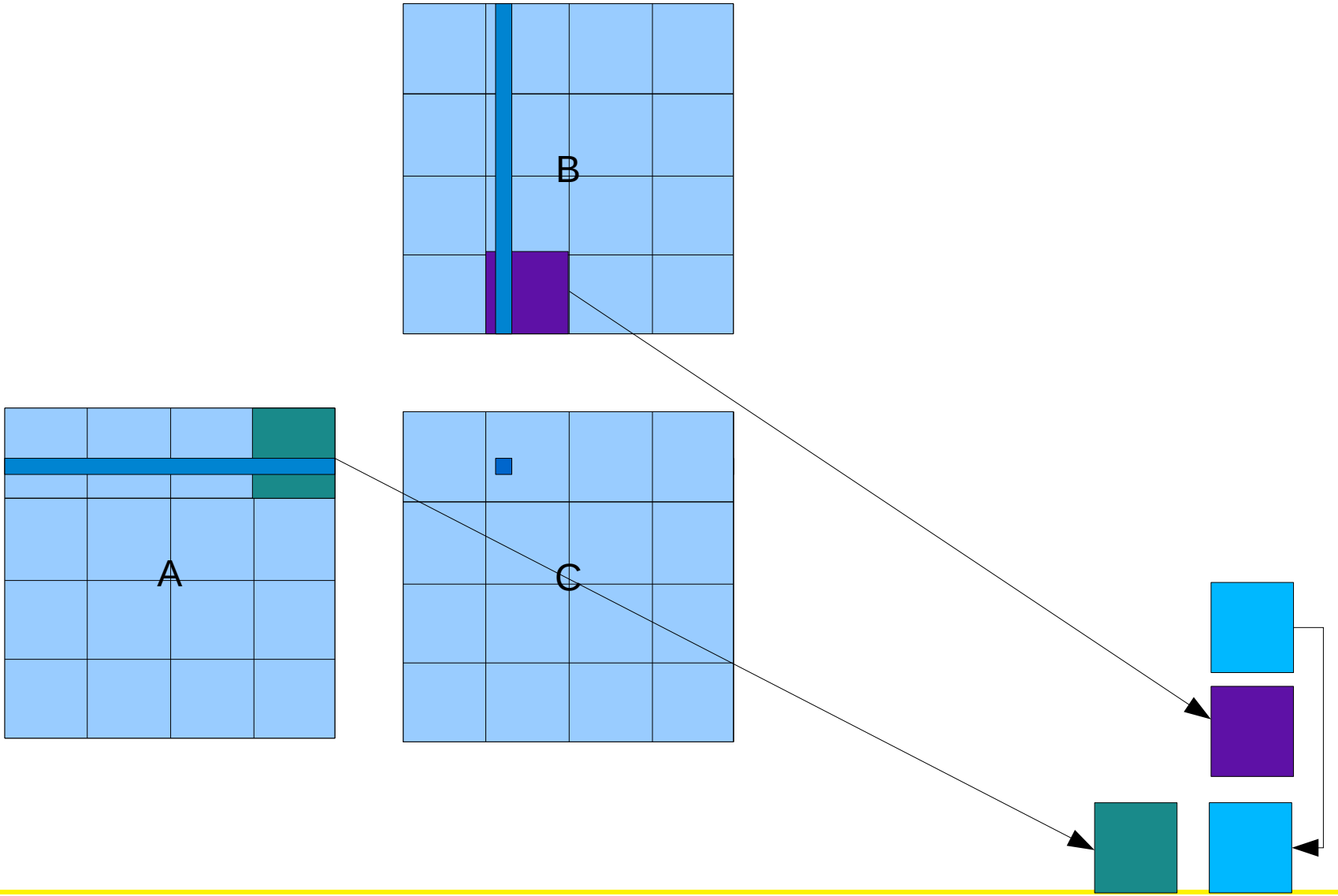
Multiplicação de matrizes



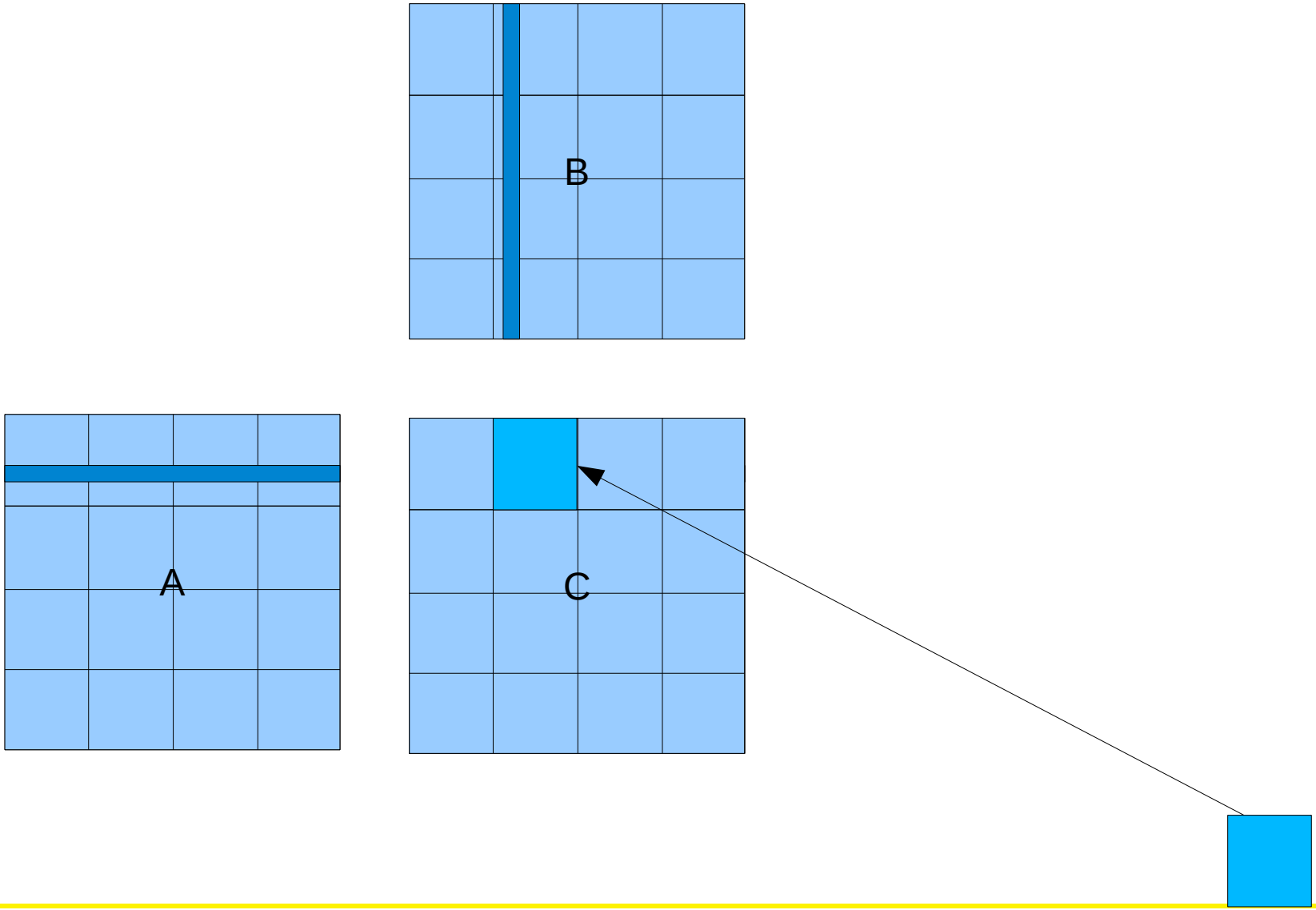
Multiplicação de matrizes



Multiplicação de matrizes



Multiplicação de matrizes



Multiplicação de matrizes

Memória *Shared*

Copiar um bloco para os caches

```
    a_cache[TINDEX(ti,tj)] = a[INDEX(i, tj + k*BW)];
    b_cache[TINDEX(ti,tj)] = b[INDEX(tj + k*BW, j)];
```

Multiplicação de matrizes

Memória *Shared*

Declarar caches

```
#define N 1024
#define BW 16
#define BLOCKS (N/BW)
#define INDEX(i,j) ((i)+((j)*N))
#define TINDEX(i,j) ((i)+((j)*BW))

__shared__ float a_cache[BW*BW];
__shared__ float b_cache[BW*BW];
```

Multiplicação de matrizes

Memória *Shared*

Copiar um bloco para os caches

Multiplicação de matrizes

Memória *Shared*

Fazer o produto parcial

Multiplicação de matrizes

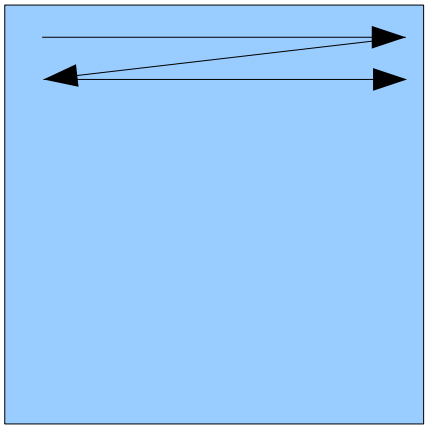
Memória *Shared*

Fazer o produto parcial

```
val += a_cache[TINDEX(ti,kk)]*b_cache[TINDEX(kk,tj)];
```

Multiplicação de matrizes

Acesso à memória sequencial



?

Multiplicação de matrizes

Acesso à memória sequencial

e se...

```
int ti = threadIdx.y;
int tj = threadIdx.x;
int i = ti + blockIdx.y*blockDim.y;
int j = tj + blockIdx.x*blockDim.x;
```

Multiplicação de matrizes

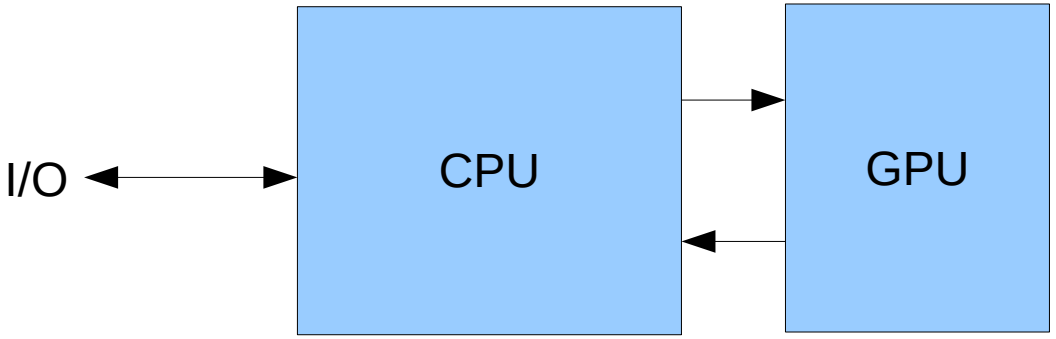
Tamanho dos blocos:

- Quanta memória *shared* usa um bloco, em função de *BW?*
  - Lembre-se: cada *float* usa 4 *bytes*
- Quanta memória tem um processador?

- Quantos blocos podem ser executados simultaneamente?

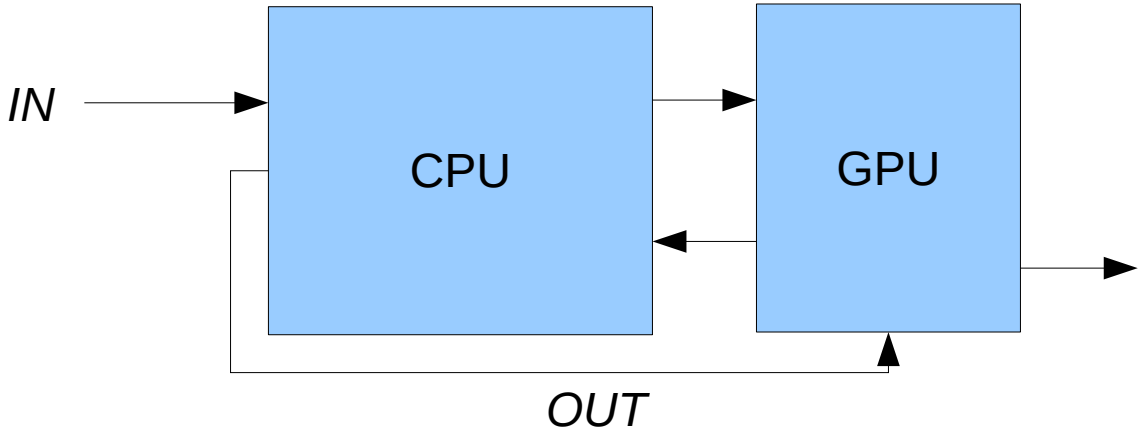
Interoperabilidade Gráfica

- Modelo de computação



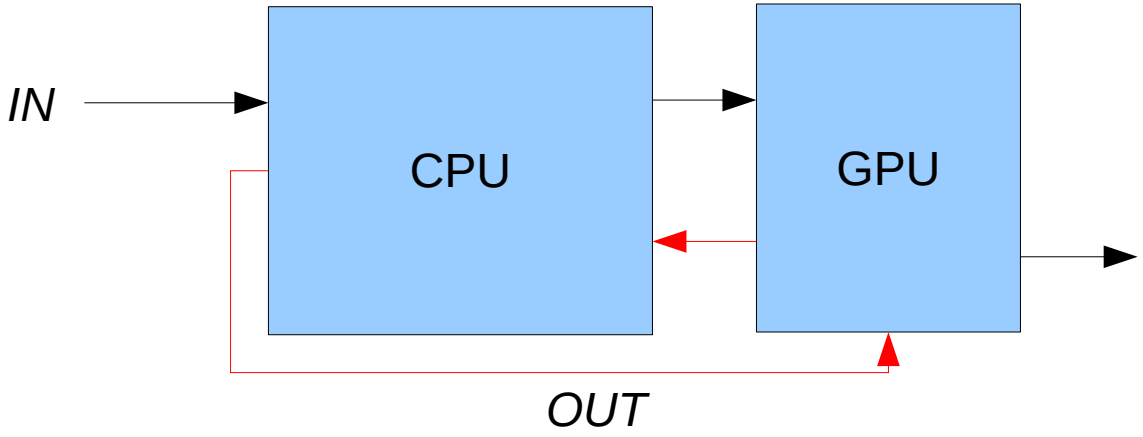
Interoperabilidade Gráfica

- Mas se o resultado final for uma imagem a ser exibida?



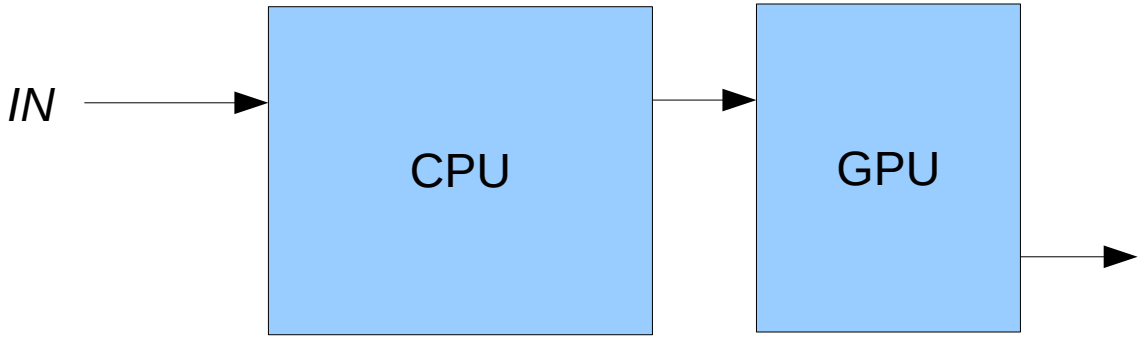
Interoperabilidade Gráfica

- Mas se o resultado final for uma imagem a ser exibida?



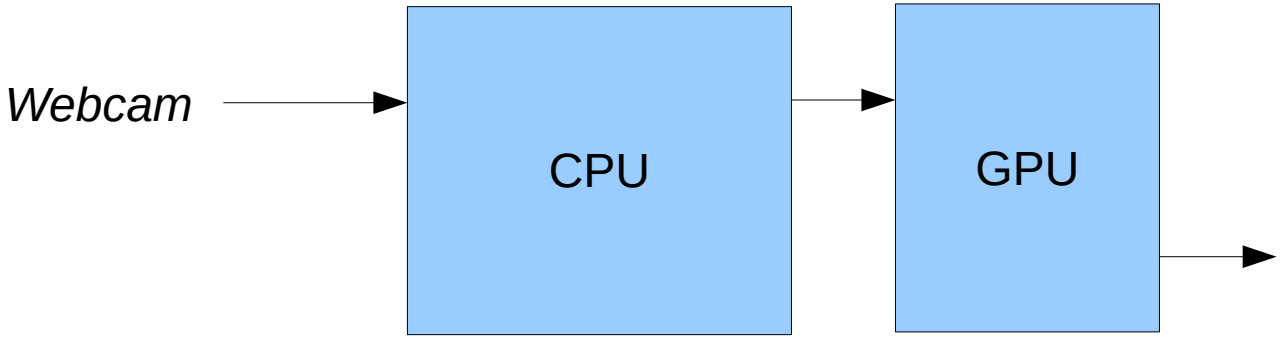
Interoperabilidade Gráfica

- Mas se o resultado final for uma imagem a ser exibida?
- A GPU também é um processador gŕafico!



Interoperabilidade Gráfica

- Exemplo: Processamento *online* de imagem



Interoperabilidade Gráfica

- Exemplo: Processamento *online* de imagem

```
void GLImage::newFrame(const unsigned char* rgbadata, int width, int length)
{
    genBuffer(pixelBuffer, rgbadata, width, length);
```

```
void GLImage::paintGL() {
    glDrawPixels(WIDTH, HEIGHT, GL_BGRA, GL_UNSIGNED_BYTE, 0);
}
```

Interoperabilidade Gráfica

- Exemplo: Processamento *online* de imagem

```
void genBuffer(GLuint buff, const unsigned char *rgbadata, int width, int heigth)
{
  cudaEvent_t start, stop;
  HANDLE_ERROR( cudaEventCreate(&start));
  HANDLE_ERROR( cudaEventCreate(&stop));

  if(width!=640 || heigth!=480) return;
  cudaGraphicsResource *resource;
  HANDLE_ERROR(cudaGraphicsGLRegisterBuffer( &resource, buff, cudaGraphicsMapFlagsNone));
  HANDLE_ERROR(cudaGraphicsMapResources( 1, &resource, NULL ) );
  uchar4 *devPtr;
  size_t  size;
  HANDLE_ERROR(cudaGraphicsResourceGetMappedPointer( (void**)&devPtr, &size, resource));

  HANDLE_ERROR(cudaEventRecord(start,0));
  HANDLE_ERROR( cudaMemcpy( devPtr, rgbadata, 4*width*heigth,
                            cudaMemcpyHostToDevice ));
  HANDLE_ERROR(cudaEventRecord(stop,0));
  HANDLE_ERROR(cudaEventSynchronize(stop));
  HANDLE_ERROR(cudaEventElapsedTime(&kerneltime, start, stop));

  HANDLE_ERROR( cudaGraphicsUnmapResources( 1, &resource, NULL ) );
}
```

Interoperabilidade Gráfica

- Exemplo: Processamento *online* de imagem

```
void genBuffer(void *resource, const unsigned char *rgbadata, int width, int heigth)
```

```
HANDLE_ERROR( cudaGraphicsMapResources( 1, (cudaGraphicsResource **)&resource, NULL ) );
  uchar4 *devPtr;
  size_t  size;
  HANDLE_ERROR(cudaGraphicsResourceGetMappedPointer( (void**)&devPtr, &size,
(cudaGraphicsResource *)resource));

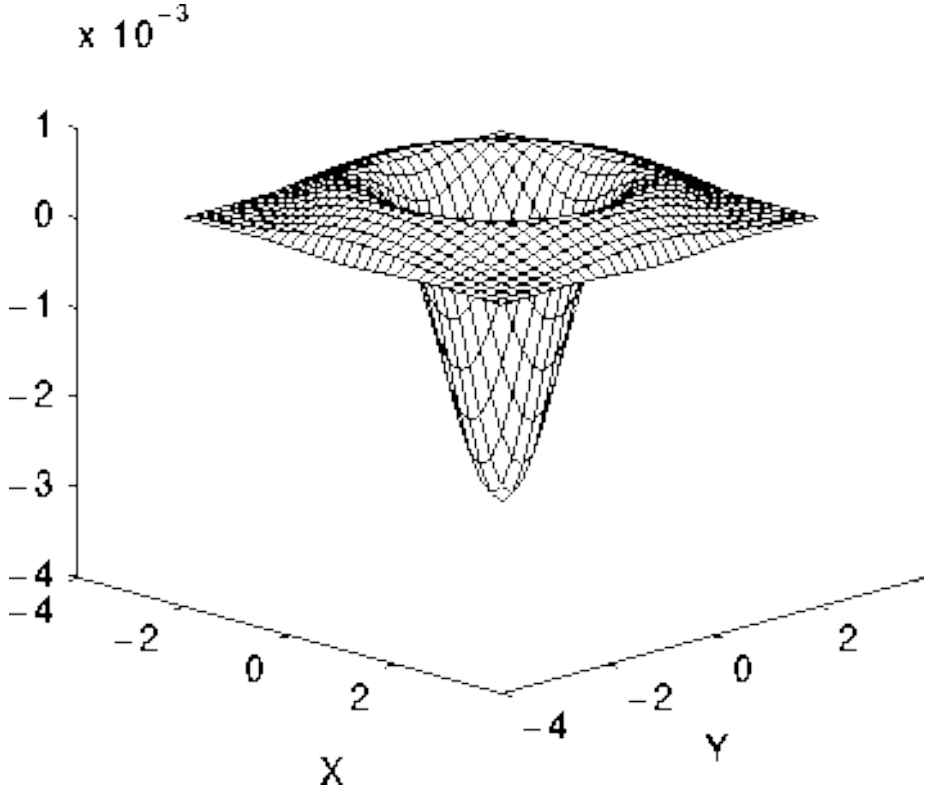
  HANDLE_ERROR(cudaEventRecord(start,0));
  HANDLE_ERROR( cudaMemcpy( devPtr, rgbadata, 4*width*heigth,
                            cudaMemcpyHostToDevice ));
  HANDLE_ERROR(cudaEventRecord(stop,0));
  HANDLE_ERROR(cudaEventSynchronize(stop));
  HANDLE_ERROR(cudaEventElapsedTime(&kerneltime, start, stop));

  HANDLE_ERROR( cudaGraphicsUnmapResources( 1, (cudaGraphicsResource **)&resource,NULL));
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem

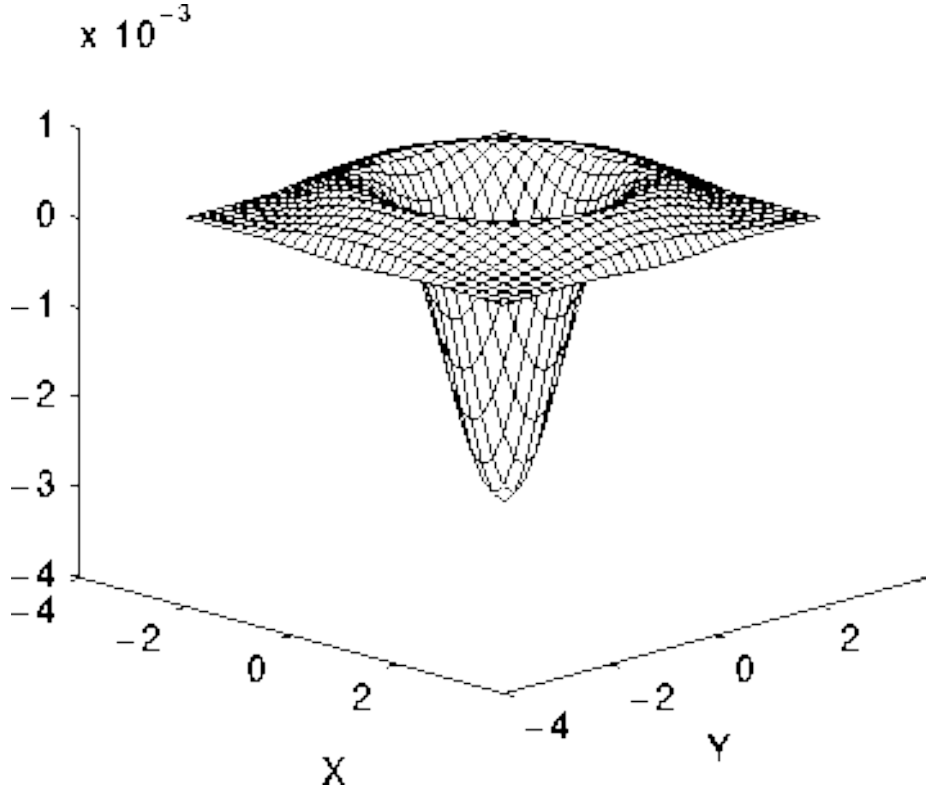
$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right]e^{-\frac{x^2+y^2}{2\sigma^2}}$$



Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem

$$(a*b)_{xy} = \sum_{ky=1}^{n} \sum_{kx=1}^{n} a_{ij} b_{(x-kx)(y-ky)}$$



Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem

```
void init()
{
```

```
    float aux[KERNEL_W][KERNEL_H];
    for(int i=0;i<KERNEL_W;i++) {
      for(int j=0;j<KERNEL_W;j++) {
     int x = i-KERNEL_W/2;
     int y = j-KERNEL_H/2;
     aux[i][j] = filtervalue(x,y);
      }
    }
    HANDLE_ERROR( cudaMalloc( (void**)&filter_kernel, sizeof(float)*KERNEL_W*KERNEL_H ) );
    HANDLE_ERROR( cudaMemcpy( filter_kernel, aux, sizeof(float)*KERNEL_W*KERNEL_H ,
                              cudaMemcpyHostToDevice ));
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem

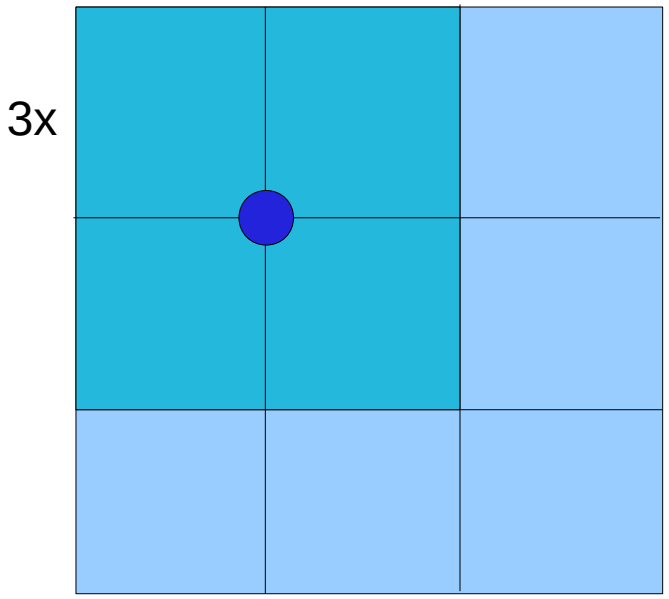
```
__global__ void kernel(bgra *out, bgra *in, float *filter_kernel)
{
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;

    if(x<16 || x >= (WIDTH-16)
      || y<16 || y >= (HEIGHT-16))
      out[INDEX(x,y)] = bgra();

    else {
      float val;
      for(int ky = y+KERNEL_MIN_Y; ky<=y+KERNEL_MAX_Y;ky++) {
        for(int kx = x+KERNEL_MIN_X; kx<=x+KERNEL_MAX_X;kx++) {
       val += getfilterkernel(x-kx,y-ky, filter_kernel)*color(in[INDEX(kx,ky)]).to_mono();
     }
      }
      out[INDEX(x,y)] = 8*val;
    }
}
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem
  - Cache local para a convolução



Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem
  - Cache local para a convolução

```
__shared__ float cache[48][48];

__global__ void kernel(bgra *out, bgra *in, float *filter_kernel)
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem
  - Cache local para a convolução

```
cache[threadIdx.y][threadIdx.x] = color(in[INDEX(x-16,y-16)]).to_mono();
cache[threadIdx.y][threadIdx.x+16] = color(in[INDEX(x,y-16)]).to_mono();
cache[threadIdx.y][threadIdx.x+32] = color(in[INDEX(x+16,y-16)]).to_mono();

cache[threadIdx.y+16][threadIdx.x] = color(in[INDEX(x-16,y)]).to_mono();
cache[threadIdx.y+16][threadIdx.x+16] = color(in[INDEX(x,y)]).to_mono();
cache[threadIdx.y+16][threadIdx.x+32] = color(in[INDEX(x+16,y)]).to_mono();

cache[threadIdx.y+32][threadIdx.x] = color(in[INDEX(x-16,y+16)]).to_mono();
cache[threadIdx.y+32][threadIdx.x+16] = color(in[INDEX(x,y+16)]).to_mono();
cache[threadIdx.y+32][threadIdx.x+32] = color(in[INDEX(x+16,y+16)]).to_mono();
__syncthreads();
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem
  - Cache local para a convolução

```
float val;
for(int ky = y+KERNEL_MIN_Y; ky<=y+KERNEL_MAX_Y;ky++) {
  for(int kx = x+KERNEL_MIN_X; kx<=x+KERNEL_MAX_X;kx++) {
 int cx = 16+kx - blockIdx.x * blockDim.x;
 int cy = 16+ky - blockIdx.y * blockDim.y;
 val += getfilterkernel(x-kx,y-ky, filter_kernel)*cache[cy][cx];
  }
}
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem
  - Filtro armazenado em **memória constante**

```
 __constant__ float filter_kernel[KERNEL_W][KERNEL_H];

__device__ float getfilterkernel(int x, int y)
{
     if(x>KERNEL_MAX_X || x < KERNEL_MIN_X || x>KERNEL_MAX_Y || x < KERNEL_MIN_Y) return 0;
     return filter_kernel[y+KERNEL_H/2][x+KERNEL_H/2];
}
```

Interoperabilidade Gráfica

- Exemplo: Filtragem em tempo real de imagem
  - Filtro armazenado em **memória constante**

```
void init()
```

```
   float aux[KERNEL_W][KERNEL_H];
   for(int i=0;i<KERNEL_W;i++) {
     for(int j=0;j<KERNEL_W;j++) {
    int x = i-KERNEL_W/2;
    int y = j-KERNEL_H/2;
    aux[i][j] = filtervalue(x,y);
     }
   }
   HANDLE_ERROR(cudaMemcpyToSymbol(filter_kernel, aux, sizeof(filter_kernel)));
```

Interoperabilidade Gráfica

- Múltiplos *buffers*

```
glGenBuffers(1, &pixelBuffer1);
glGenBuffers(1, &pixelBuffer2);

glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pixelBuffer1);
glBufferData(GL_PIXEL_UNPACK_BUFFER, 4* WIDTH * HEIGHT, NULL,
          GL_DYNAMIC_DRAW);

glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pixelBuffer2);
glBufferData(GL_PIXEL_UNPACK_BUFFER, 4* WIDTH * HEIGHT, NULL,
          GL_DYNAMIC_DRAW);

resourceBuffer1 = registerCudaResource(pixelBuffer1);
resourceBuffer2 = registerCudaResource(pixelBuffer2);
```

Interoperabilidade Gráfica

- Múltiplos *buffers*

```
 void GLImage::newFrame(QImage *img)
{
    genBuffer(resourceBuffer2, resourceBuffer1, img->constBits(), img->width(),
           img->height());
    averageTime += kerneltime;
    timeSamples++;
    if(timeSamples>30) {
      this->setWindowTitle(QString::number(averageTime/timeSamples) + "ms");
      averageTime = 0;
      timeSamples = 0;
    }
    this->repaint();
    delete img;
}
```

Interoperabilidade Gráfica

- Múltiplos *buffers*

```
void GLImage::paintGL() {
     glPixelZoom(1,-1);
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pixelBuffer1);
    glRasterPos2f(-1.0,1.0);
    glDrawPixels(WIDTH, HEIGHT, GL_BGRA, GL_UNSIGNED_BYTE, 0);

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pixelBuffer2);
    glRasterPos2f(-1.0,0.0);
    glDrawPixels(WIDTH, HEIGHT, GL_BGRA, GL_UNSIGNED_BYTE, 0);
}
```

Interoperabilidade Gráfica

- Múltiplos *buffers*

```
void genBuffer(void *resource1, void *resource2, const unsigned char *rgbadata, int width,
     int heigth)
```

```
HANDLE_ERROR( cudaGraphicsMapResources( 1, (cudaGraphicsResource **)&resource1, NULL ) );
  bgra* devPtr1;
  size_t  size;
  HANDLE_ERROR(cudaGraphicsResourceGetMappedPointer( (void**)&devPtr1, &size,
(cudaGraphicsResource *)resource1));

  HANDLE_ERROR(cudaEventRecord(start,0));
  HANDLE_ERROR( cudaMemcpy( gpubuff, rgbadata, 4*width*heigth,
                            cudaMemcpyHostToDevice ));

  dim3    grids(WIDTH/16,HEIGHT/16);
  dim3    threads(16,16);


  kernel<<<grids,threads>>>( devPtr1 , gpubuff);

  HANDLE_ERROR( cudaGraphicsUnmapResources( 1, (cudaGraphicsResource **)&resource1,
           NULL ) );
```

Interoperabilidade Gráfica

- Múltiplos *buffers*

```
void genBuffer(void *resource1, void *resource2, const unsigned char *rgbadata, int width,
     int heigth)
```

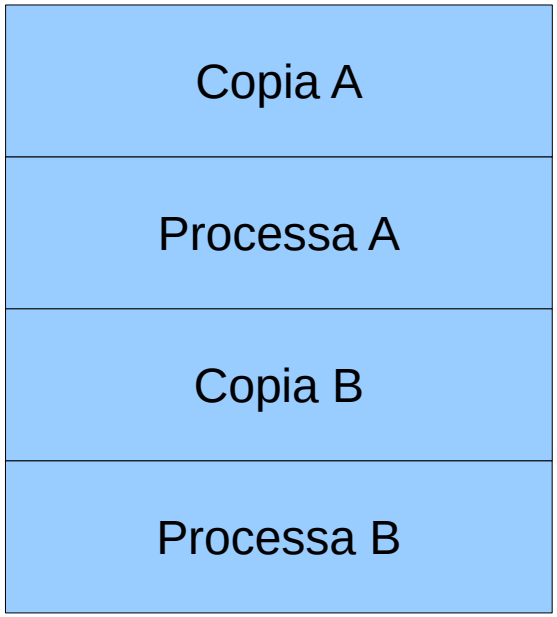
```
HANDLE_ERROR( cudaGraphicsUnmapResources( 1, (cudaGraphicsResource **)&resource1, NULL ) );


  HANDLE_ERROR( cudaGraphicsMapResources( 1, (cudaGraphicsResource **)&resource2, NULL ) );
  void *devPtr2;
  HANDLE_ERROR(cudaGraphicsResourceGetMappedPointer( (void**)&devPtr2, &size,
(cudaGraphicsResource *)resource2));

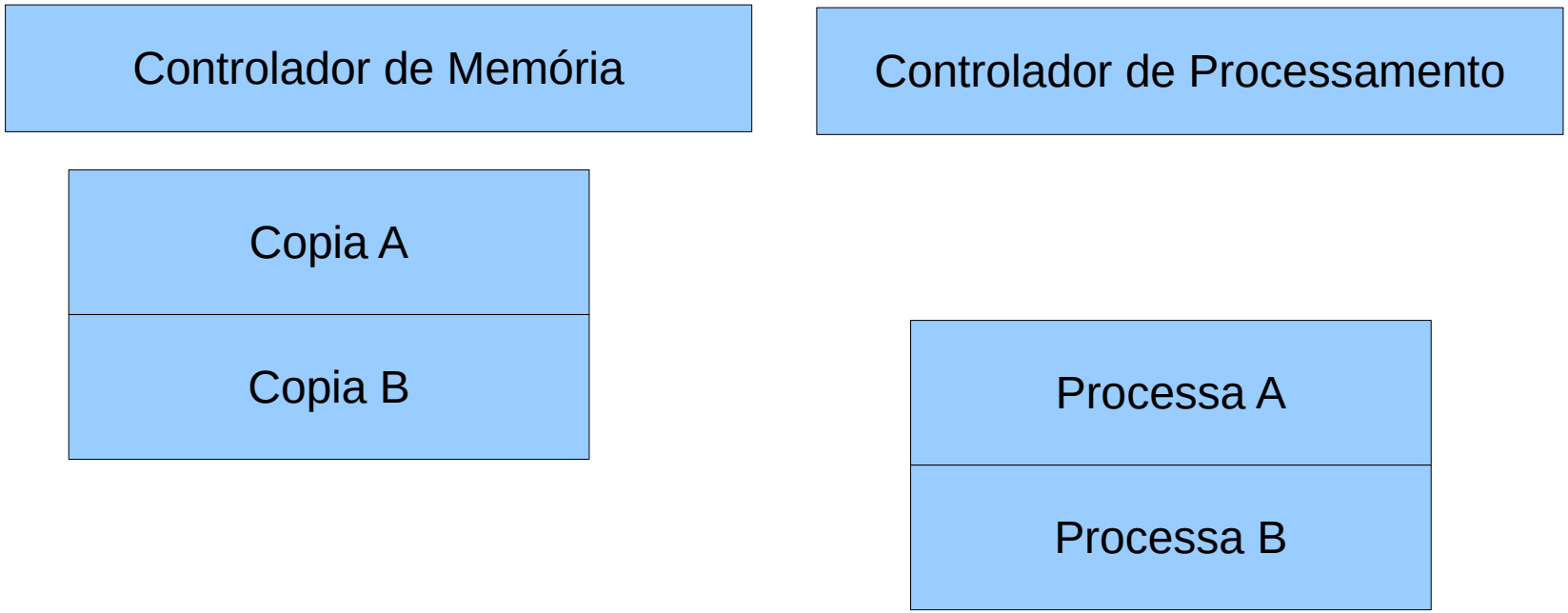
  HANDLE_ERROR( cudaMemcpy( devPtr2, rgbadata, 4*width*heigth,
                            cudaMemcpyHostToDevice ));

  HANDLE_ERROR( cudaGraphicsUnmapResources( 1, (cudaGraphicsResource **)&resource2,
NULL ) );
```

Fluxos múltiplos!



Fluxos múltiplos!



Fluxos múltiplos!

```
HANDLE_ERROR(cudaStreamCreate(&streamA));
HANDLE_ERROR(cudaStreamCreate(&streamB));
```

```
HANDLE_ERROR( cudaMemcpyAsync( gpubuff, rgbadata, 4*width*heigth,
                               cudaMemcpyHostToDevice, streamA ));
kernel<<<grids,threads, 0, streamA>>>( devPtr1, gpubuff);
HANDLE_ERROR( cudaMemcpyAsync( devPtr2, rgbadata, 4*width*heigth,
                               cudaMemcpyHostToDevice, streamB));

HANDLE_ERROR(cudaStreamSynchronize(streamA));
HANDLE_ERROR(cudaStreamSynchronize(streamB));
```