Introdução ao Modelo de Programação CUDA

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente - LaSDPC

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP

Primórdios...

- Até fim dos anos 90 tínhamos renderização gráfica em computadores por meio de controladores VGA (*Video Graphics Array*)
  - Basicamente um controlador de memória e uma interface para exibição de vídeo
  - Associado a uma memória DRAM
- Nos anos 2000 surgiram as GPUs (*Graphics Processing Units*) para substituir controladores VGA
  - Permitiam renderização de imagens

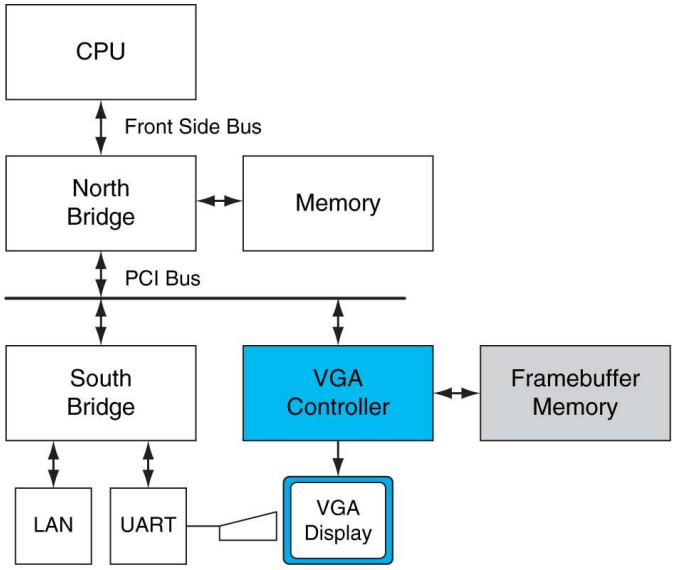


FIGURE A.2.1 Historical PC. VGA controller drives graphics display from framebuffer memory. Copyright © 2009 Elsevier, Inc. All rights reserved.

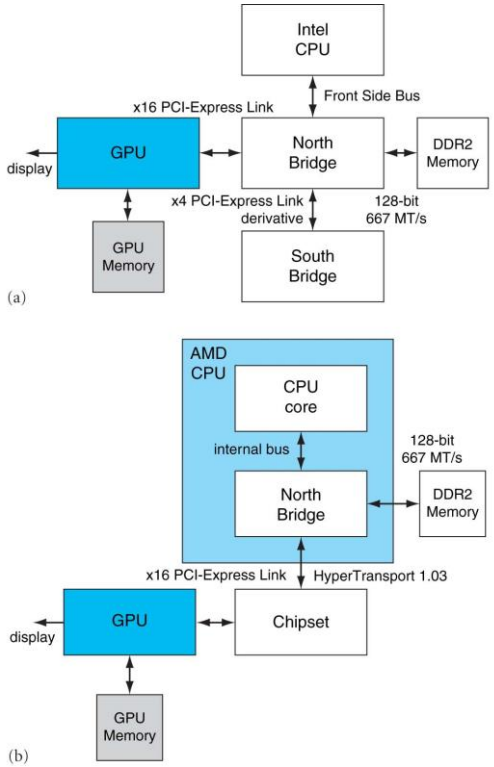


FIGURE A.2.2 Contemporary PCs with Intel and AMD CPUs. See Chapter 6 for an explanation of the components and interconnects in this figure. Copyright © 2009 Elsevier, Inc. All rights reserved.

Patterson & Hennessy (2014)

Evolução das GPUs

- Novas tecnologias foram incorporadas às GPUs
  - Renderizações de melhor qualidade,
  - Novos efeitos visuais
  - Melhor precisão em cálculos complexos relacionados ao processamento de imagens

- **GPUs** tornaram-se **programáveis** para dar mais flexibilidade aos códigos executados
  - Como subproduto permitiu o uso de dispositivos gráficos para computação de alto desempenho em outros nichos de aplicação, além das gráficas
  - Nasce o conceito de GPGPU (*General Purpose Graphics Processing Unit*)

- GPUs são dispositivos (*devices*) com milhares de processadores (*cores*) especializados
  - Agrupados em vetores (*arrays*)
  - Com alto grau de paralelismo e escalabilidade
  - Com foco em aumentar a vazão de instruções (*throughput*) quando comparadas a CPUs escalares

- Possuem características SIMD, porém não são SIMD puras
  - Implementam o paradigma **SIMT** (*Single Instruction Multiple Thread*), onde uma única instrução é executada por múltiplas *threads*
  - Oferecem suporte para *multithreading* maciço, com replicação de instr por diferentes *threads*
  - *SIMT flexibiliza máquinas SIMD por não estarem restritas à execução de uma única instrução, mas à execução simultânea de múltiplas threads, majoritariamente executando a mesma instrução*

Evolução das GPUs

- GPUs têm uma sofisticada estrutura de memória para dar suporte à alta vazão de instruções

- **Registradores**
  - Localizados nos SPs e armazenam o estado das *threads* em execução no SM

- **Memória local à *thread***
  - Memórias SRAM, privadas às *threads*

- **Memória constante**
  - Memória somente de leitura pelas *threads* de um SM

- **Memória compartilhada**
  - Memória SRAM usada pelas *threads* de um SM. Acesso bem mais rápido que a global

- **Memória de textura**
  - Memória DRAM que armazena coordenadas e outros dados referentes à textura das imagens
  - Têm o apoio de caches de textura

- **Memória global**
  - Memória DRAM acessada por quaisquer *threads* de qualquer SM

- Há também outras memórias intermediárias e caches a depender da arquitetura para redução da latência de acesso

CPU vs GPU

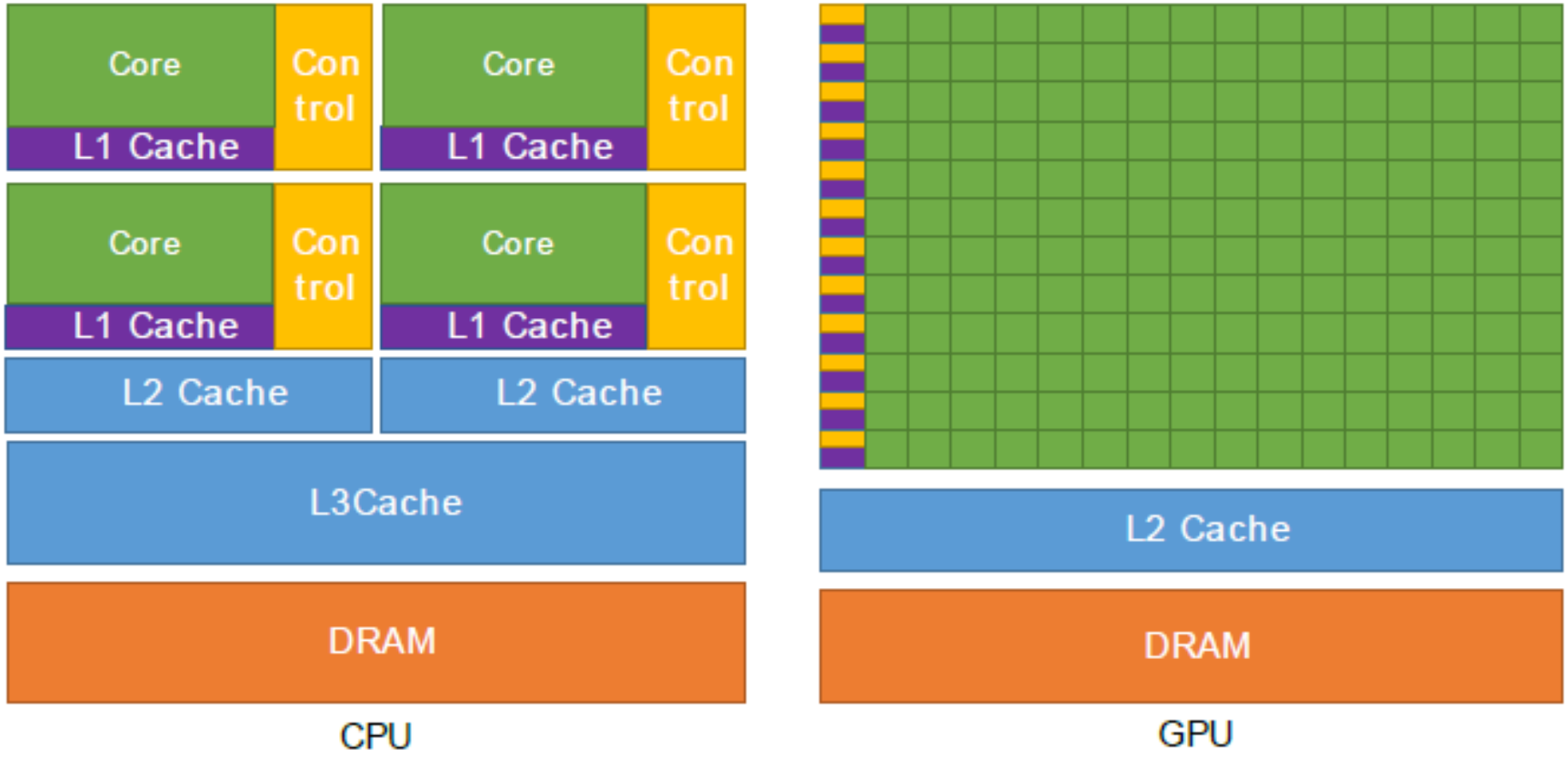


Figure 1 The GPU Devotes More Transistors to Data Processing

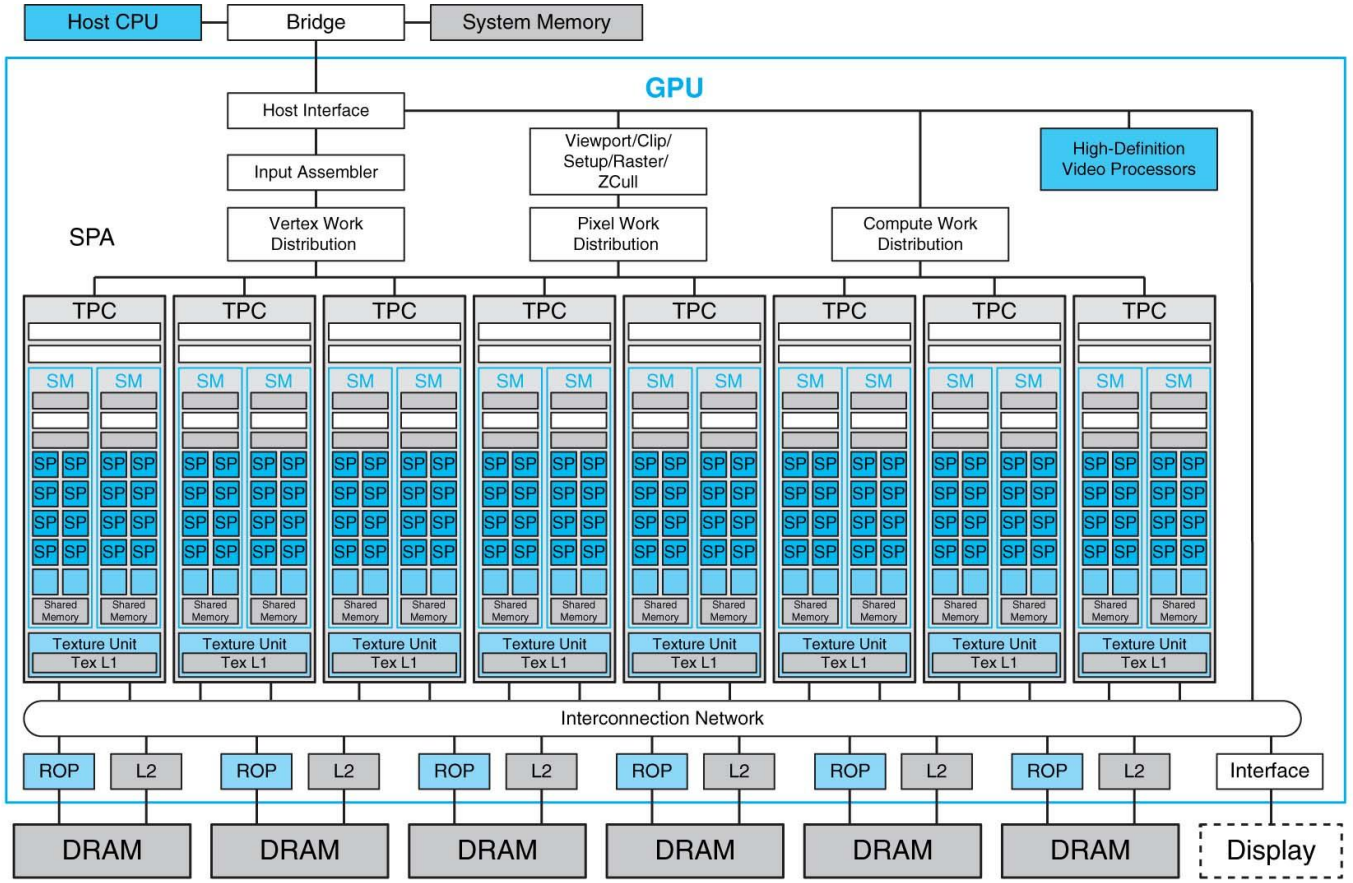


FIGURE A.7.1 NVIDIA Tesla unified graphics and computing GPU architecture. This GeForce 8800 has 128 streaming processor (SP) cores in 16 streaming multiprocessors (SM), arranged in eight texture/processor clusters (TPC). The processors connect with six 64-bit-wide DRAM partitions via an interconnection network. Other GPUs implementing the Tesla architecture vary the number of SP cores, SMs, DRAM partitions, and other units.

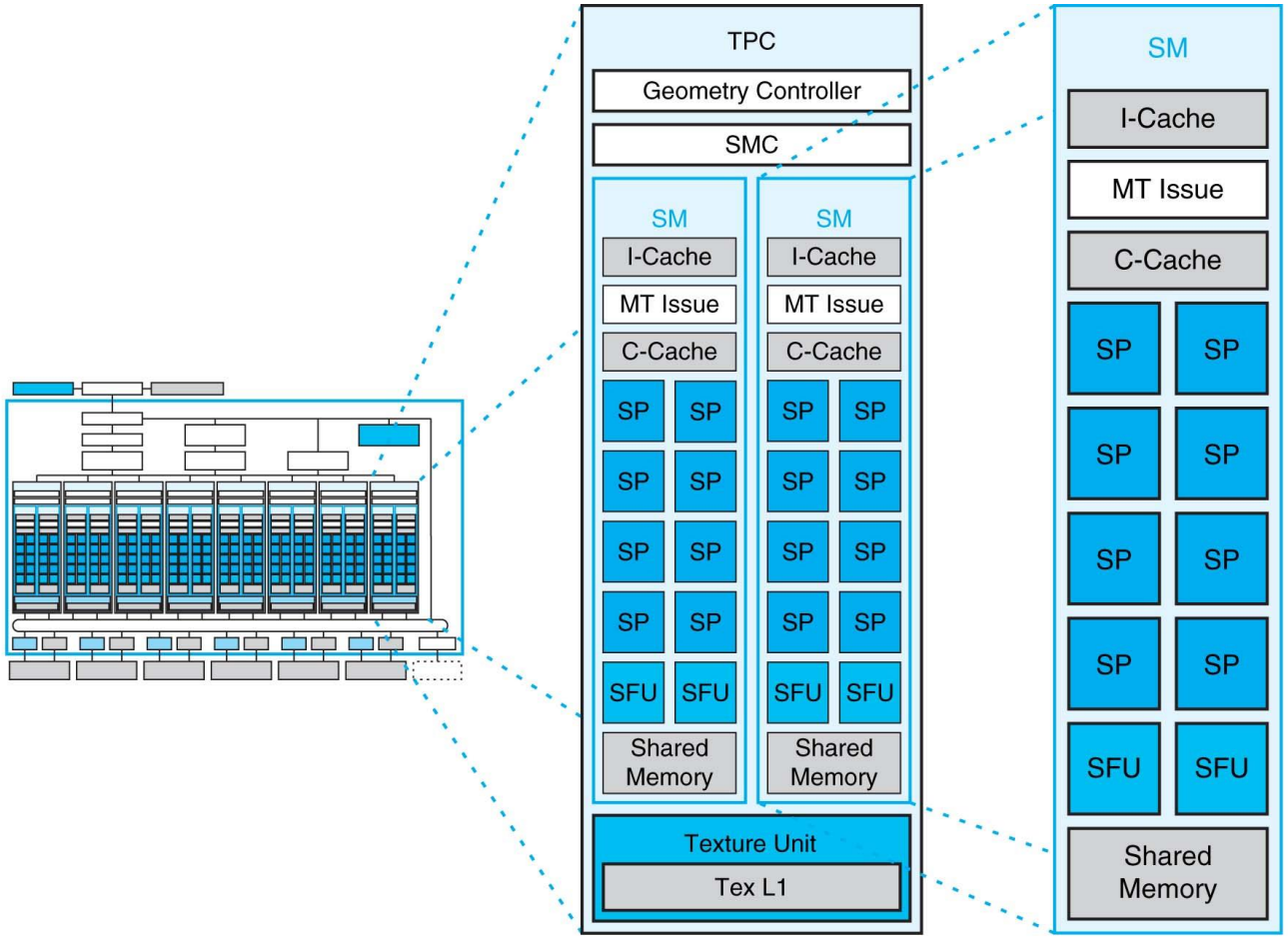


FIGURE A.7.2 Texture/processor cluster (TPC) and a streaming multiprocessor (SM). Each SM has eight streaming processor (SP) cores, two SFUs, and a shared memory.

Modelo de Programação CUDA

- CUDA (*Compute-Unified Device Architecture*) é um modelo de programação paralela
  - Atua como uma extensão de C/C++ e Fortran para programar dispositivos heterogêneos
    - CPU + processadores específicos (como a GPU)
  - Desenvolvido pela NVDIA em 2006

- Uma *thread* CUDA é um vetor de instruções SIMD a serem executadas sequencialmente
  - Esse vetor é replicado e opera sobre dados diferentes

- *Warp* é um conjunto de 32 *threads* CUDA, de um bloco de *threads*, que são executadas simultaneamente
  - Escalonador de warps no hardware decide por atribuir instruções de um *warp* aos SPs disponíveis em um SM

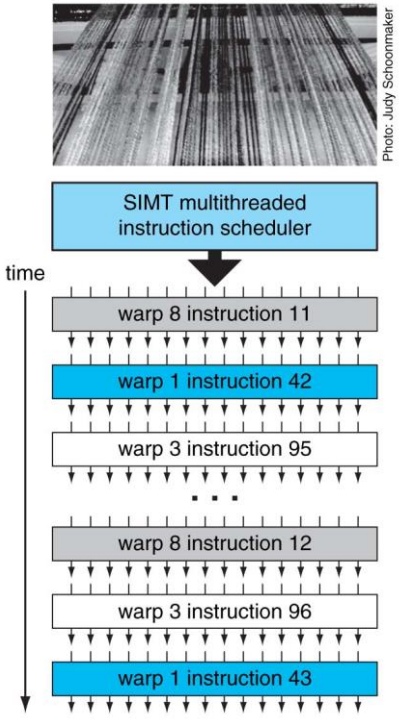


FIGURE A.4.2 SIMT multithreaded warp scheduling. The scheduler selects a ready warp and issues an instruction synchronously to the parallel threads composing the warp. Because warps are independent, the scheduler may select a different warp each time. Copyright © 2009 Elsevier, Inc. All rights reserved.

Patterson & Hennessy (2014)

Terminologia Básica para CUDA

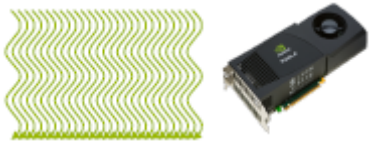
- Terminologia
  - ***Host***
    - CPU e sua hierarquia de memória (*host memory*)

Host

  - ***Device***
    - GPU e suas memórias (*device memory*)

Device

  - ***Kernel***
    - Função que é replicada na GPU como múltiplas *threads*



Estrutura Geral de um Programa C/CUDA

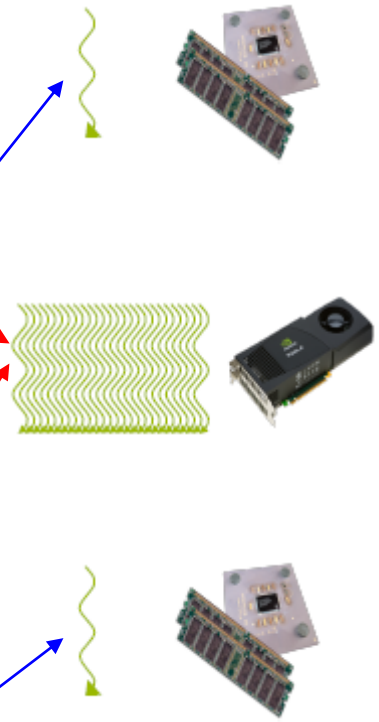
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RDS 3
#define BLCK 16

__global__ void stencil_1d(int *in, int *out)    {
      __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
      int gindex = threadIdx.x + blockIdx.x * blockDim.x;
      int lindex = threadIdx.x + RADIUS;
      // Read input elements into shared memory
      temp[lindex] = in[gindex];
      if (threadIdx.x < RADIUS)          {
            temp[lindex - RADIUS] = in[gindex - RADIUS];
            temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
      }

      // Synchronize (ensure all the data is available)
      __syncthreads();

      // Apply the stencil
      int result = 0;
      for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
         result += temp[lindex + offset];

         // Store the result
         out[gindex] = result;
}

void fill_ints(int *x, int n)    {
      fill_n(x, n, 1);
}

int main(void)    {
      int *in, *out; // host copies of a, b, c
      int *d_in, *d_out; // device copies of a, b, c
      int size = (N + 2*RADIUS) * sizeof(int);

      // Alloc space for host copies and setup values
      in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
      out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

      // Alloc space for device copies
      cudaMalloc((void **)&d_in, size);
      cudaMalloc((void **)&d_out, size);

      // Copy to device
      cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

      // Launch stencil_1d() kernel on GPU
      stencil_1d <<<N/BLCK,BLCK>>>(d_in + RDS, d_out + RDS);

      // Copy result back to host
      cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

      // Cleanup
      free(in); free(out);
      cudaFree(d_in); cudaFree(d_out);
      return 0;
}
```

Função paralela replicada na GPU (*device*) na forma de múltiplas *threads*
É o kernel.

Código sequencial executado na CPU (*host*)

Lançamento das *threads* na GPU

Código seq exec na CPU (*host*)



Stringhini (2017)

Compilar & Executar Programas C/CUDA

- Detalhes operacionais com programas C/CUDA
  - O programas fonte têm **extensão *.cu***
  - Inclua este arquivo de cabeçalho
    - ***#include <cuda.h>***
  - Para compilar use
    - ***nvcc fonte.cu –o binário***

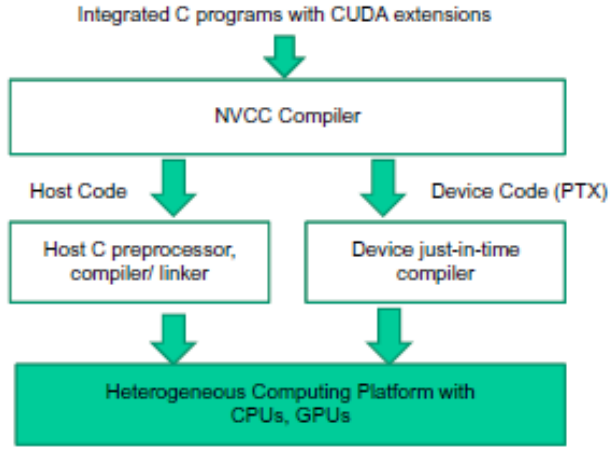


FIGURE 3.2

Overview of the compilation process of a CUDA program.

  - Execute o programa compilado diretamente do *shell*
    - ***binario <enter>***
    - O processo começará a executar como uma *thread* no *host* (CPU)

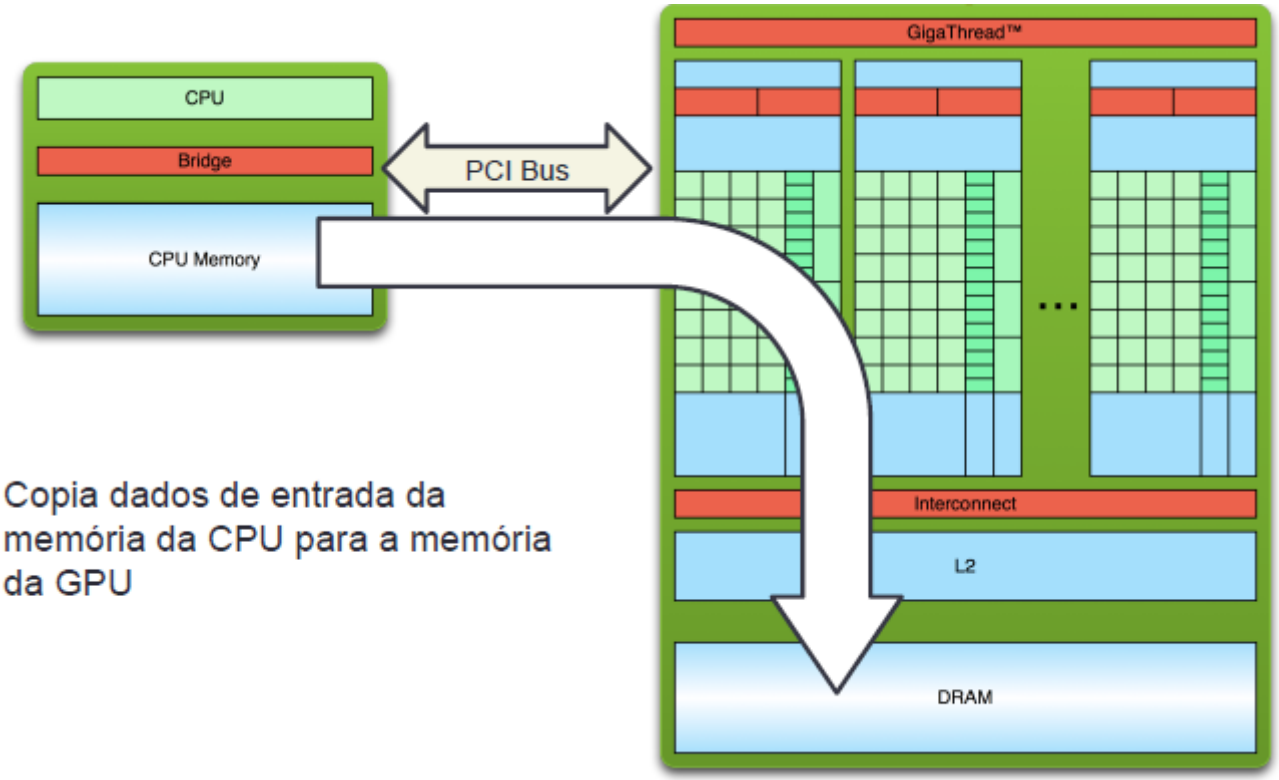
Kirk & Hwu (2013)

Aspectos Básicos de Memórias em CPU e GPU

- Gerenciamento de Memória
  - As memórias do *host* e do *device* são usualmente separadas
    - *Device* usa ponteiros para a memória da GPU
      - Devem ser passados de/para a memória do *host*
    - *Host* usa ponteiros para a memória da CPU
      - Podem ser passados de/para o *device*
  - API CUDA para a gerência de memória do *device*
    - *cudaMalloc(), cudaFree(), cudaMemcpy()*
    - Análogas às funções C *malloc()*, *free()* e *memcpy()*
  - Memórias da CPU e GPU podem ser vistas como unificadas
    - Veremos essas memórias mais à frente

Passos Usuais e Básicos de um Programa CUDA

- Um Fluxo de Processamento Simples

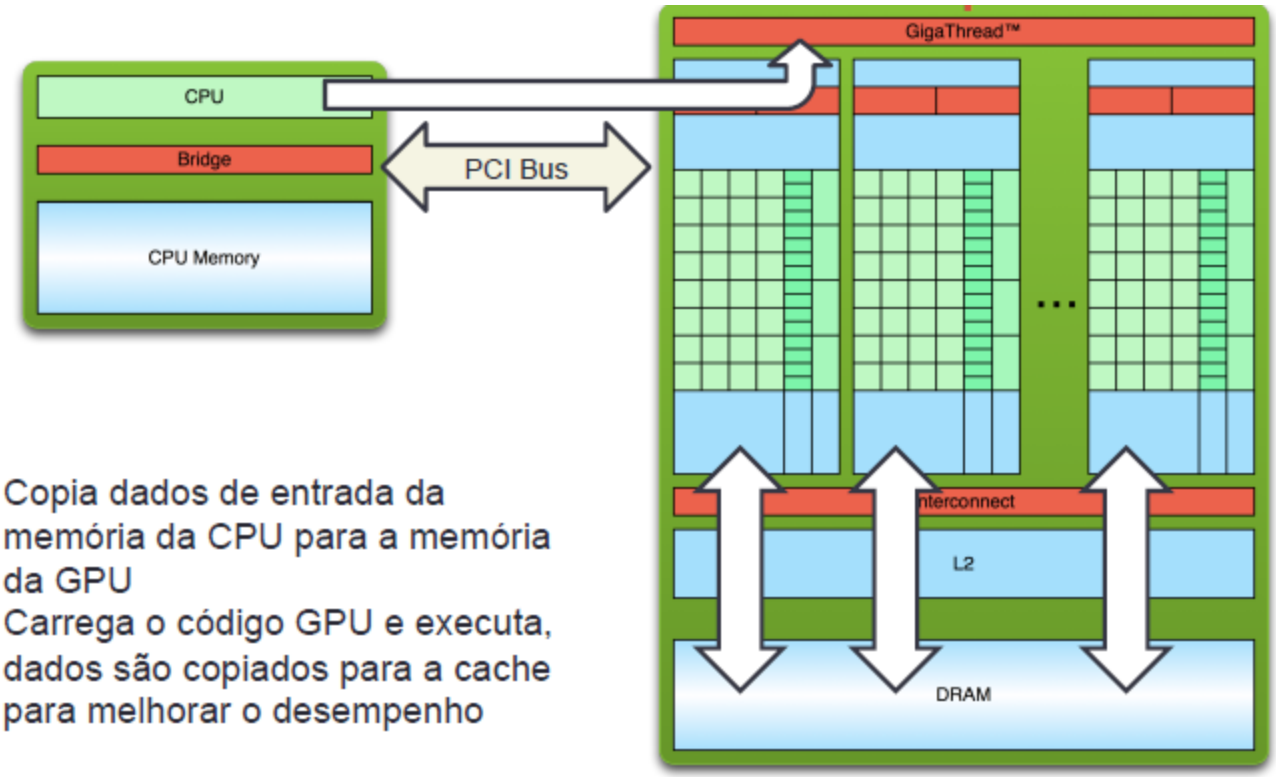


1. Copia dados de entrada da memória da CPU para a memória da GPU

Stringhini (2017)

Passos Usuais e Básicos de um Programa CUDA

- Um Fluxo de Processamento Simples

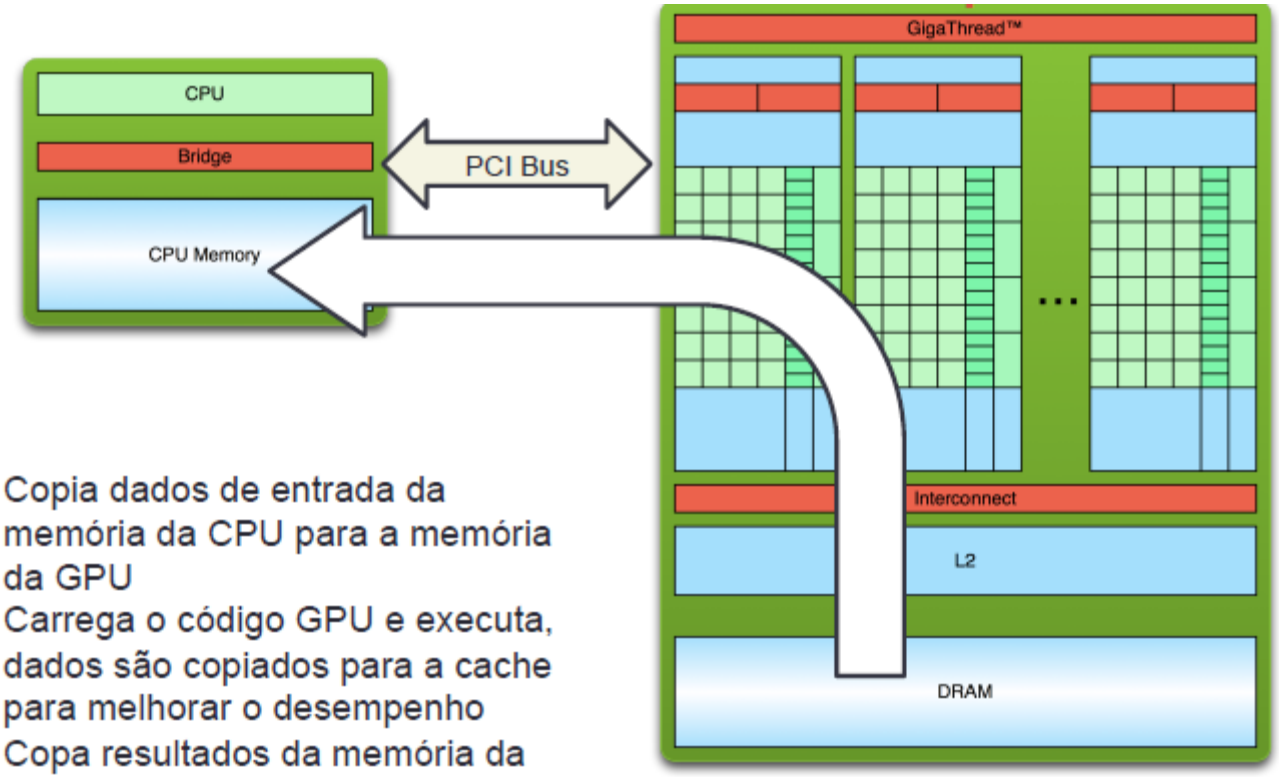


1. Copia dados de entrada da memória da CPU para a memória da GPU
2. Carrega o código GPU e executa, dados são copiados para a cache para melhorar o desempenho

Stringhini (2017)

Passos Usuais e Básicos de um Programa CUDA

- Um Fluxo de Processamento Simples



1. Copia dados de entrada da memória da CPU para a memória da GPU
2. Carrega o código GPU e executa, dados são copiados para a cache para melhorar o desempenho
3. Copa resultados da memória da GPU para a memória da CPU

Stringhini (2017)

Passos Usuais e Básicos de um Programa CUDA

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
   int size = n* sizeof(float);
   float *A_d, *B_d, *C_d;
   …
1. // Allocate device memory for A, B, and C
    // copy A and B to device memory

2. // Kernel launch code – to have the device
    // to perform the actual vector addition

3. // copy C from the device memory
    // Free device vectors
}
```

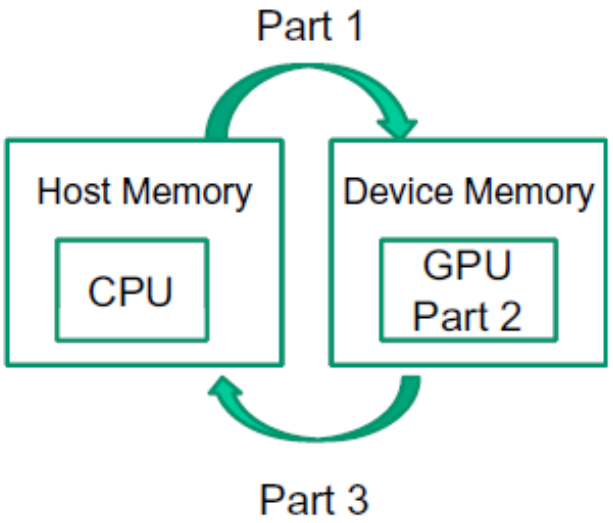


FIGURE 3.5

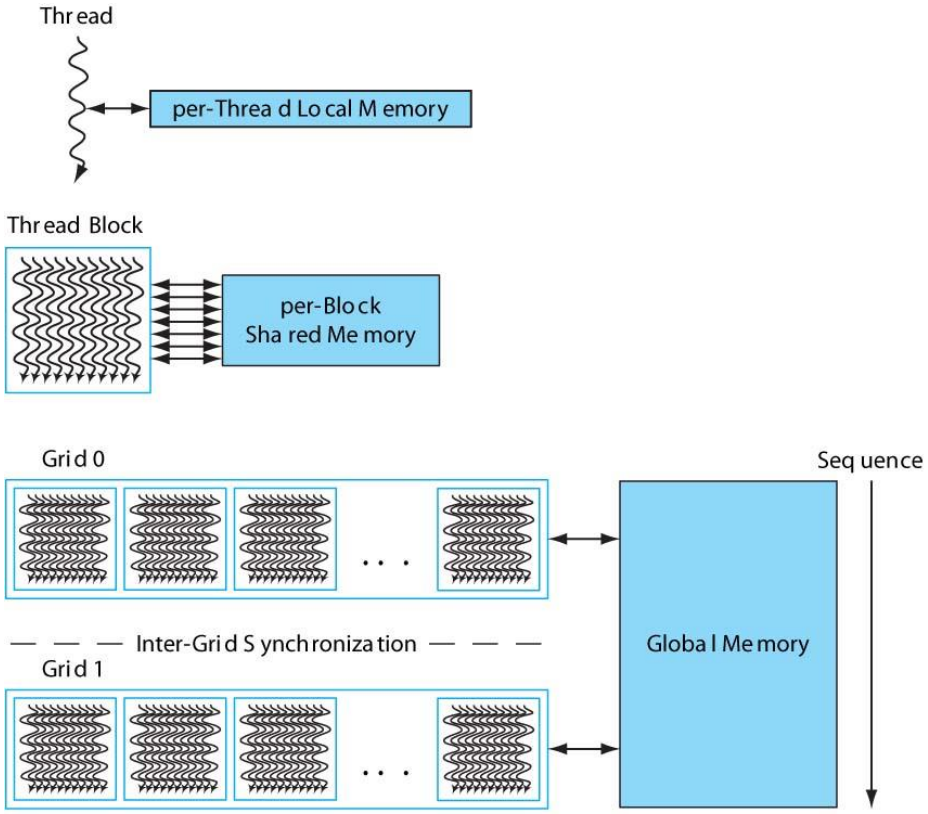
Outline of a revised `vecAdd()` function that moves the work to a device.

Kirk & Hwu (2013)

Hierarquia de *Threads* em CUDA

- Threads CUDA são agrupadas em um modelo hierárquico
  - Níveis permitem diferentes interações entre *threads* e acessos a diferentes memórias
    - ***Threads***
      - Organizadas em blocos de *threads*
      - Possuem memória local à *thread* e ao bloco a que pertencem
      - Até 1024 *threads* por bloco
    - **Blocos**
      - Organizados em grades de blocos
      - Agrupam diferentes *threads*
      - Permitem que suas *threads* compartilhem memória
      - *Threads* de um mesmo bloco são executadas em um mesmo SM (*Streaming Multiprocessor*)
    - **Grades (*grids*)**
      - Agrupam diferentes blocos de *threads*
      - Pemitem que *threads* de diferentes blocos usem a memória global do *device*
      - *Threads* de blocos diferentes de uma grade são executadas por diferentes SMs
      - Podem haver mais grades em execução em um dado instante

Hierarquia de *Threads* em CUDA



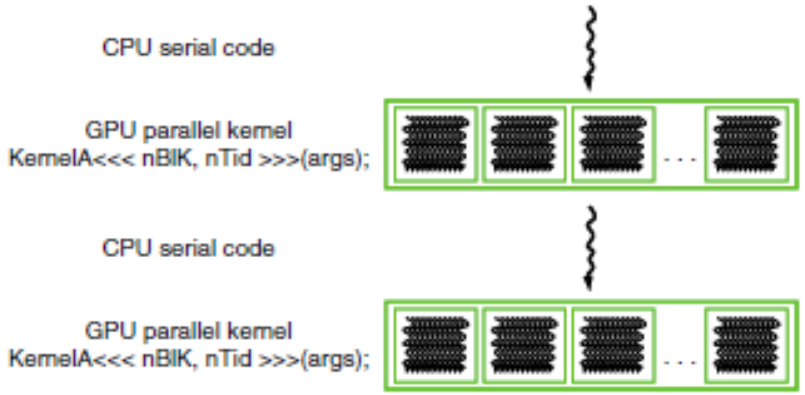


FIGURE 3.3

Execution of a CUDA program.

Patterson & Hennessy (2014)
Kirk & Hwu (2013)

Lançamento de *Threads* em Paralelo na GPU

- Relembrando: *kernel* é uma função executada por múltiplas *threads* no *device*
  - Possuem modificadores de acesso para orientar a compilação
    - ***__global__*** chamado pelo *host* e executado no *device*
    - ***__device__*** chamado pelo *device* e executado no *device*
    - ***__host__*** chamado pelo *host* e executado no *host* (*default*)
  - Exemplo:

```
__global__ void funcaoKernel(tipo_1 parametro_1,....,tipo_n parametro_n)
{
        //Bloco de código da função
}
```

  - O valor de retorno do *kernel* deve ser *void*
  - Para invocar (ou lançar) um *kernel*

funcaoKernel<<<NumeroDeBlocos,NumeroDeThreads>>>(arg1,arg2,....,argn)

onde: ***NumeroDeBlocos*** indica quantos blocos há na grade
NumeroDeThreads indica quantas *threads* há por bloco

Lançamento de *Threads* em Paralelo na GPU

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

FIGURE 3.12

CUDA C keywords for function declaration.

Kirk & Hwu (2013)

Lançamento de *Threads* em Paralelo na GPU

- Um exemplo de código: *Hello World* ☺

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

__global__ void hello()
{
   printf("Hello world\n");
}

int main(int argc,char **argv)
{
   int blocos = atoi(argv[1]);
   int threads = atoi(argv[2]);

   hello<<<blocos,threads>>>();

   cudaDeviceSynchronize();
   return(0);
}
```

Lançamento de *Threads* em Paralelo na GPU

- *Threads* CUDA podem ser identificadas em até 03 dimensões (x, y, z)

- Associa-se um novo tipo de dado, o ***dim3***

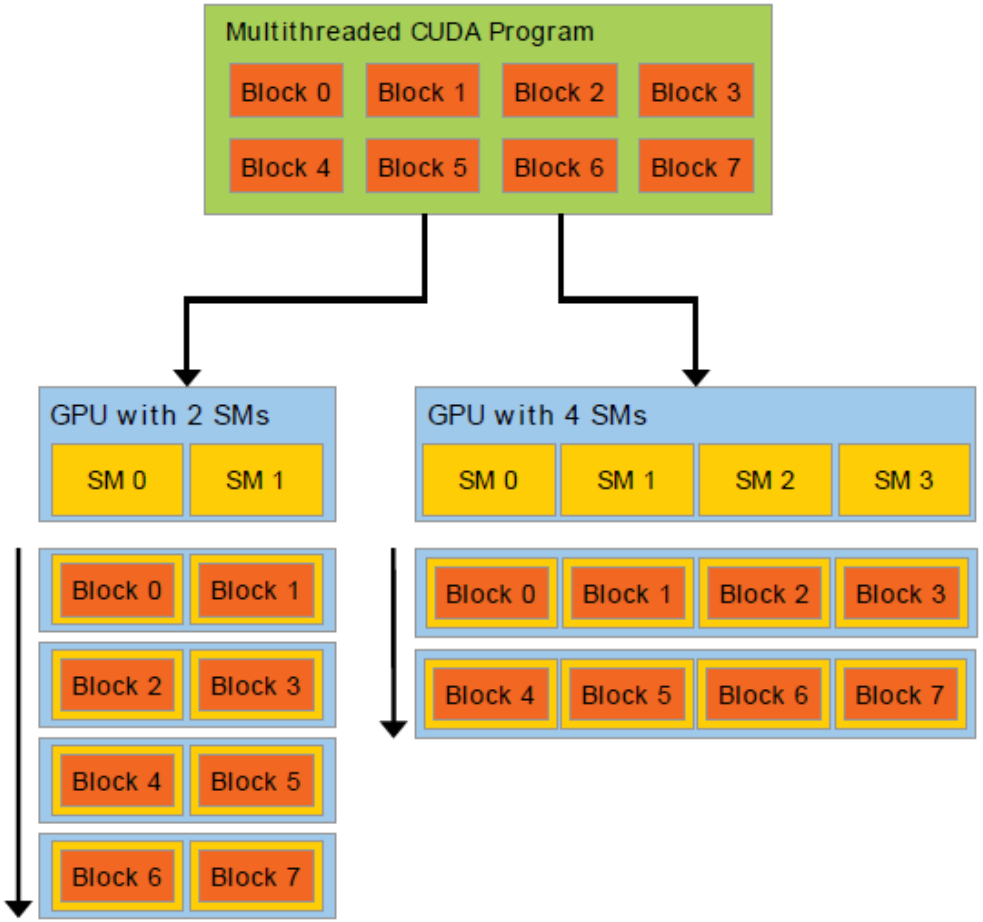
  dim3 variável(dimx, dimy, dimz);

- O uso de ***dim3*** não é obrigatório (não foi usado no exemplo anterior do *Hello World*)
  - Neste caso apenas a dimensão ***x*** foi passada explicitamente
  - Neste caso também, as dimensões ***y*** e ***z*** são 1 por padrão

- O tipo ***dim3*** pode ser usado tanto para a quantidade de blocos quanto para *threads*

- *Exemplos:*

```
dim3 dimGrid(128,1,1);
dim3 dimBloco(256,1,1);
nucleo<<<dimGrid, dimBlock>>>(size);
```



A GPU is built around an array of Streaming Multiprocessors (SMs) (see Hardware Implementation for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Figure 3 Automatic Scalability

NVIDIA-PG (2019)

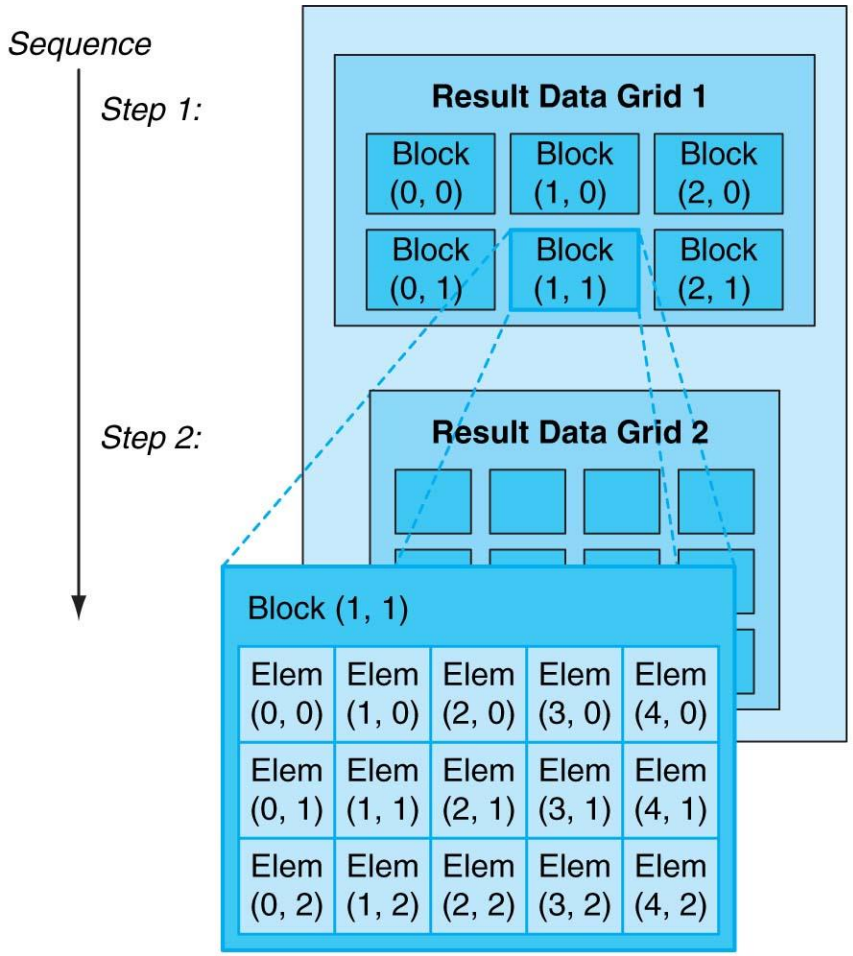


FIGURE A.3.3 Decomposing result data into a grid of blocks of elements to be computed in parallel.

Patterson & Hennessy (2014)

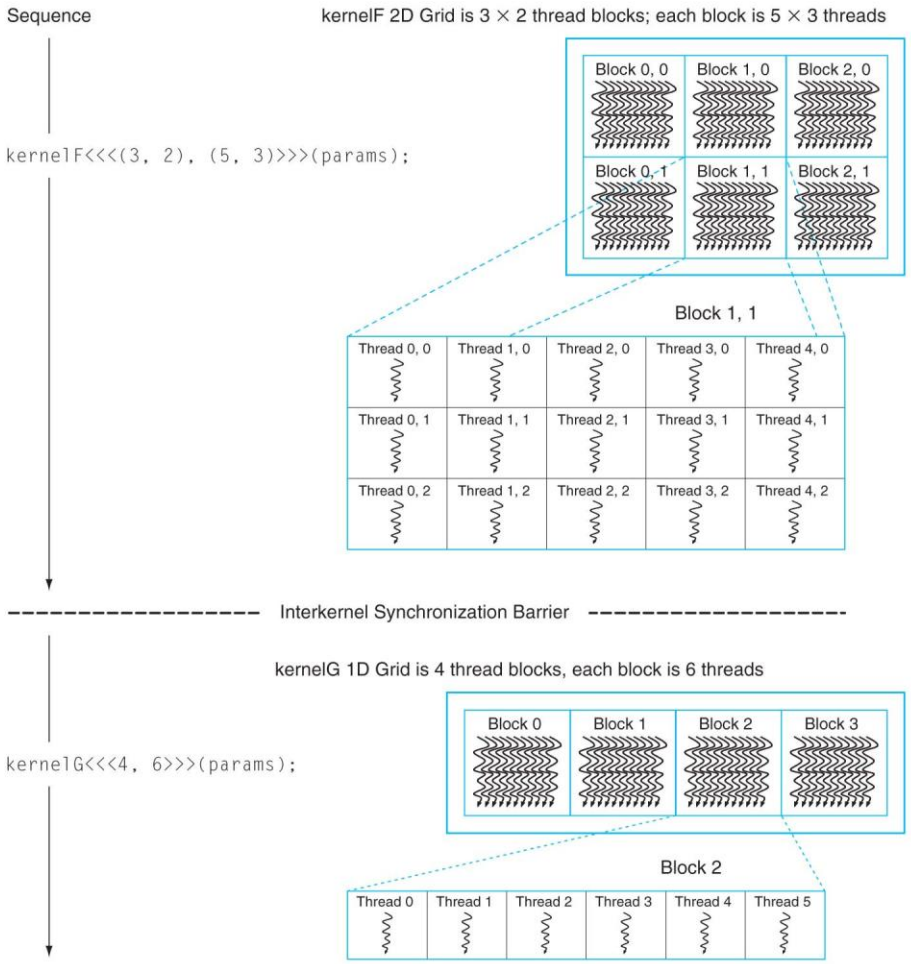


FIGURE A.3.6 Sequence of kernel *F* instantiated on a 2D grid of 2D thread blocks, an interkernel synchronization barrier, followed by kernel *G* on a 1D grid of 1D thread blocks.

Patterson & Hennessy (2014)

Variáveis pré-definidas em CUDA

- As dimensões podem ser recuperadas no *kernel* com **variáveis já pré-definidas**
  - ***gridDim{.x, .y ou .z}*** e ***blockDim{.x, .y ou .z}***

- O identificador da *thread* é dado pela variável (**também pré-definida)**
  - ***threadIdx{.x, .y ou .z}***
  - A numeração começa em 0 (zero) e é relativa ao bloco que a *thread* pertence

- O identificador do bloco é dado pela variável (**também pré-definida**)
  - ***blockIdx{.x, .y ou .z}***
  - A numeração começa em 0 (zero) e é relativa à grade que o bloco pertence

- A variável pré-definida ***warpSize*** mostra a quantidade de *threads* do *warp*

Exemplo de Variáveis Pré-Definidas em CUDA

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

__global__ void hello( )    {
    int indice_local = threadIdx.x;
    int indice_global = blockDim.x * blockIdx.x + threadIdx.x;
    int indice_bloco=blockIdx.x;
    int tamanho_grid=gridDim.x;

    printf("Hello from thread: índice global %d, índice local: %d, bloco: %d, tamanho do
grid: %d\n", indice_global, indice_local,indice_bloco,tamanho_grid);
}

int main(int argc,char **argv)    {
    int blocos=atoi(argv[1]);
    int threads=atoi(argv[2]);

    hello<<<blocos,threads>>>( );

    cudaDeviceSynchronize();
}
```

Computing y = ax + y with a serial loop:

```
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing y = ax + y in parallel using CUDA:

```
__global__
void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n )  y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

FIGURE A.3.4 Sequential code (top) in C versus parallel code (bottom) in CUDA for SAXPY (see Chapter 7). CUDA parallel threads replace the C serial loop—each thread computes the same result as one loop iteration. The parallel code computes *n* results with *n* threads organized in blocks of 256 threads.

Patterson & Hennessy (2014)

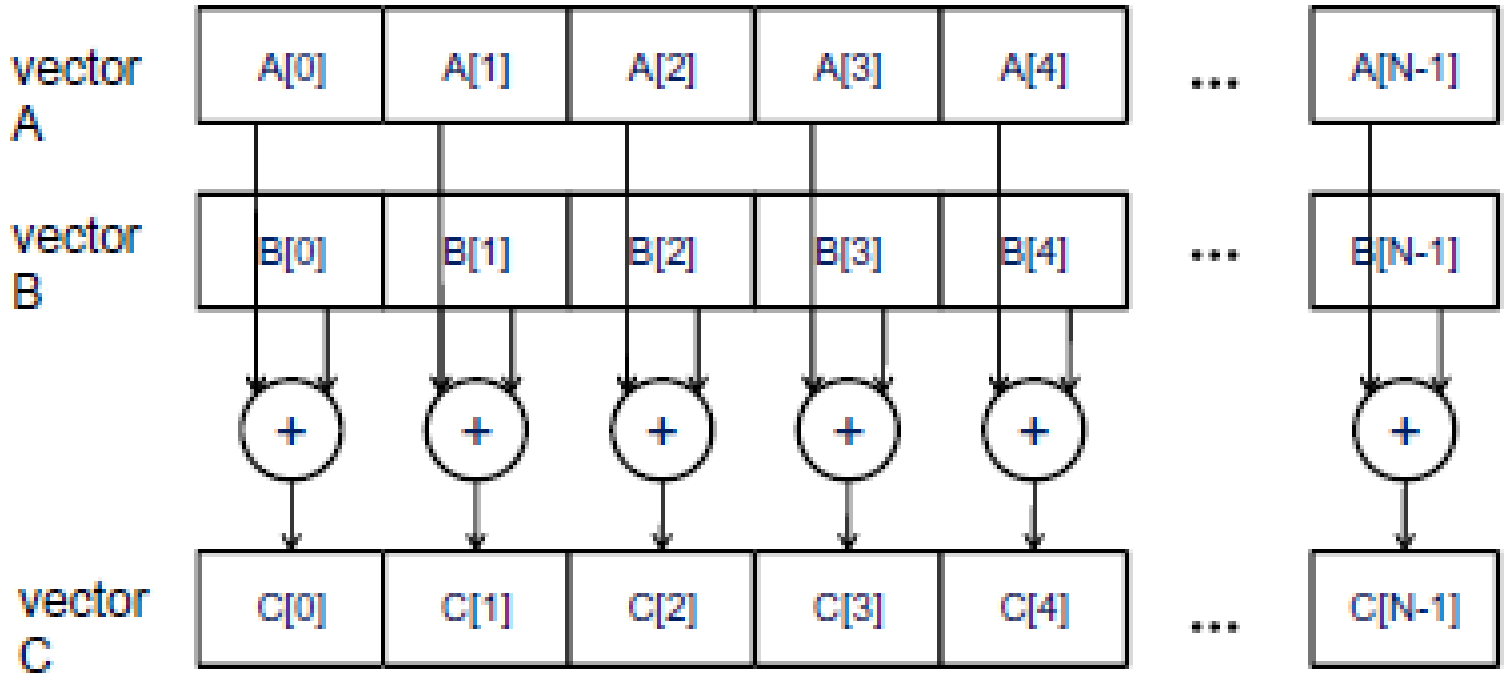


FIGURE 3.1

Data parallelism in vector addition.

Kirk & Hwu (2013)

```
// Compute vector sum h_C = h_A+h_B
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  for (i = 0; i < n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements each
    …
    vecAdd(h_A, h_B, h_C, N);
}
```

FIGURE 3.4

A simple traditional vector addition C code example.

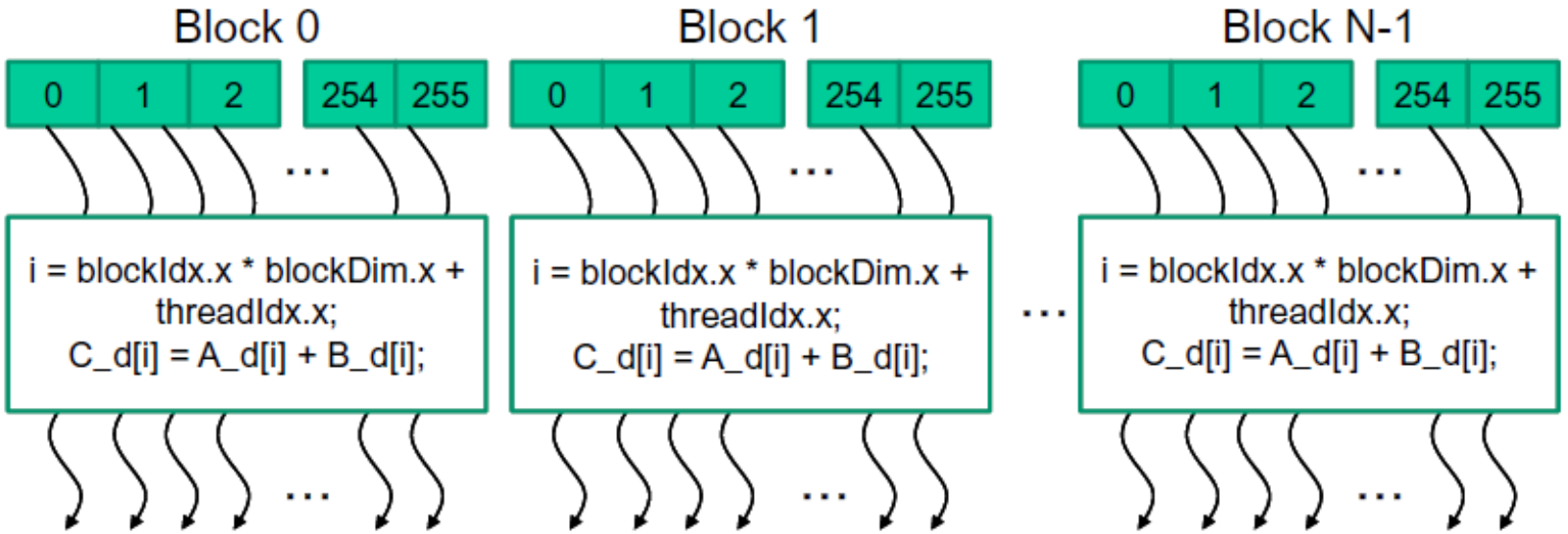


FIGURE 3.10

All threads in a grid execute the same kernel code.

Kirk & Hwu (2013)

Breve Descrição dos Exemplos de Código CUDA

- Exemplo 01 - *hello world from GPU*
  - Gera 10 blocos com 01 *thread* em cada que imprimem msg com ***blockId.x***

- Exemplo 02 – get properties
  - Imprime propriedades da versão CUDA e das GPUs disponíveis

- *Exemplo 03 – add two integers*
  - Soma dois inteiros no *kernel* e devolve o resultado
  - Exemplifica *cudaMalloc( )*

- Exemplo 04 – *add vectors t threads*
  - Soma dois vetores em paralelo
  - Exemplifica alocação, cópia e outras características de várias *threads* em um bloco

- Exemplo 05 – *add vectors b blocks*
  - Soma dois vetores em paralelo
  - Exemplifica alocação, cópia e outras características de vários blocos em um bloco

- Exemplo 06 – *add vectors b blocks & t threads*
  - Soma dois vetores em paralelo
  - Exemplifica alocação várias *threads* em vários blocos

Referências

Patterson, D.A., Hennessy, J.L. Computer organization and design : the hardware/software interface, 5th ed., Elsevier, Amsterdam, 2014,

Kirk, D.B., Hwu, W.W., Programming Massively Parallel Processors: a hands-on approach. 2nd ed., Morgan Kaufman, NVIDIA, 2013.

NVIDIA-PG (2019), Cuda C Programming: design guide, PG-02829-001_v10.1, May, 2019, Cap 01 (Introduction) e Cap 02 (Programming Model).

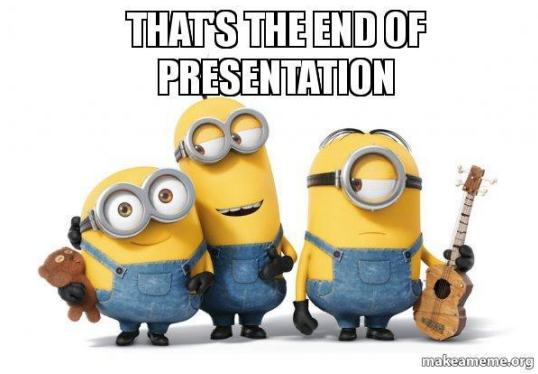
CUDA RUNTIME API, Api Reference Manual, NVIDIA, July 2019.

Stringhini, Denise. Desenvolvimento de Aplicações em CUDA. Slides apresentados na ERAD-SP 2017. São Carlos/SP. Disponível em http://eradsp2017.lasdpc.icmc.usp.br/desenvolvimento-de-aplicaoes-em-cuda/ Último acesso em 22/11/2020.

Introdução ao Modelo de Programação CUDA

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente - LaSDPC



CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP