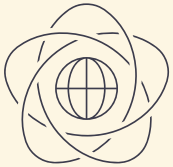


Faculty of Mathematics and Information Science

WARSAW UNIVERSITY OF TECHNOLOGY

Graphic Processors in Computational Applications

Part 1 – Introduction

dr inż. Krzysztof Kaczmarski

2021

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

Materiały sponsorowane przez:

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny

Goals for today:

- Understand course passing requirements
- Get basic knowledge on GPU programming

Semester Schedule

GPU and modern HPC

Introduction to CUDA and GPGPU
- Threads and Processes
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting
- Example

Lectures

Technical part:

1. GPU threads basics
   Process/Thread/Kernel, Host/Device
2. Memory management
   Global/Local/Shared/Registers/Constant
3. Threads synchronization
4. Advanced memory management
5. Multiple GPU - HPC
6. Advanced parallel execution problems
7. Inter-warp communication
8. Thrust API

Lectures

Algorithms:

1. Model of vector processing
2. Parallel scalability models
3. Prefix-sums
4. Parallel sorting
5. Optimal matrix multiplication
6. Particle interactions

Obligatory Laboratories

Semester Schedule

1	Tutorial: Play in the playground – choose your toys
2	Tutorial: Can you reduce? (3p)
3	Tutorial: Touch a fractal border (3p)
4	Tutorial: Trust in Thrust (3p)
5-9	Project 1 (40-60p)
10-14	Project 2 (40-60p)

Projects

Grading I

- Choose two projects from the list:
  - A (easy): 40 points
  - B (moderate): 60 points
- https://e.mini.pw.edu.pl/en/course_details/5379
- You must report progress every two weeks.
- Deadline for the projects: the last week of the semester.

Projects

Grading II

- If a project contains no mistakes it gets 100% of the possible points.
- There are penalty points for misuse of GPU concepts:
  - -10% : processor occupancy not achieved or too few threads running
  - -10% : memory allocation or deallocation problems
  - -10% : AoS if SaA is possible
  - -5% : shared memory conflicts
  - -5% : ugly code, no comments, mess in files
  - -5% : no makefile (cmake is ok)

The most powerful computers use GPU devices

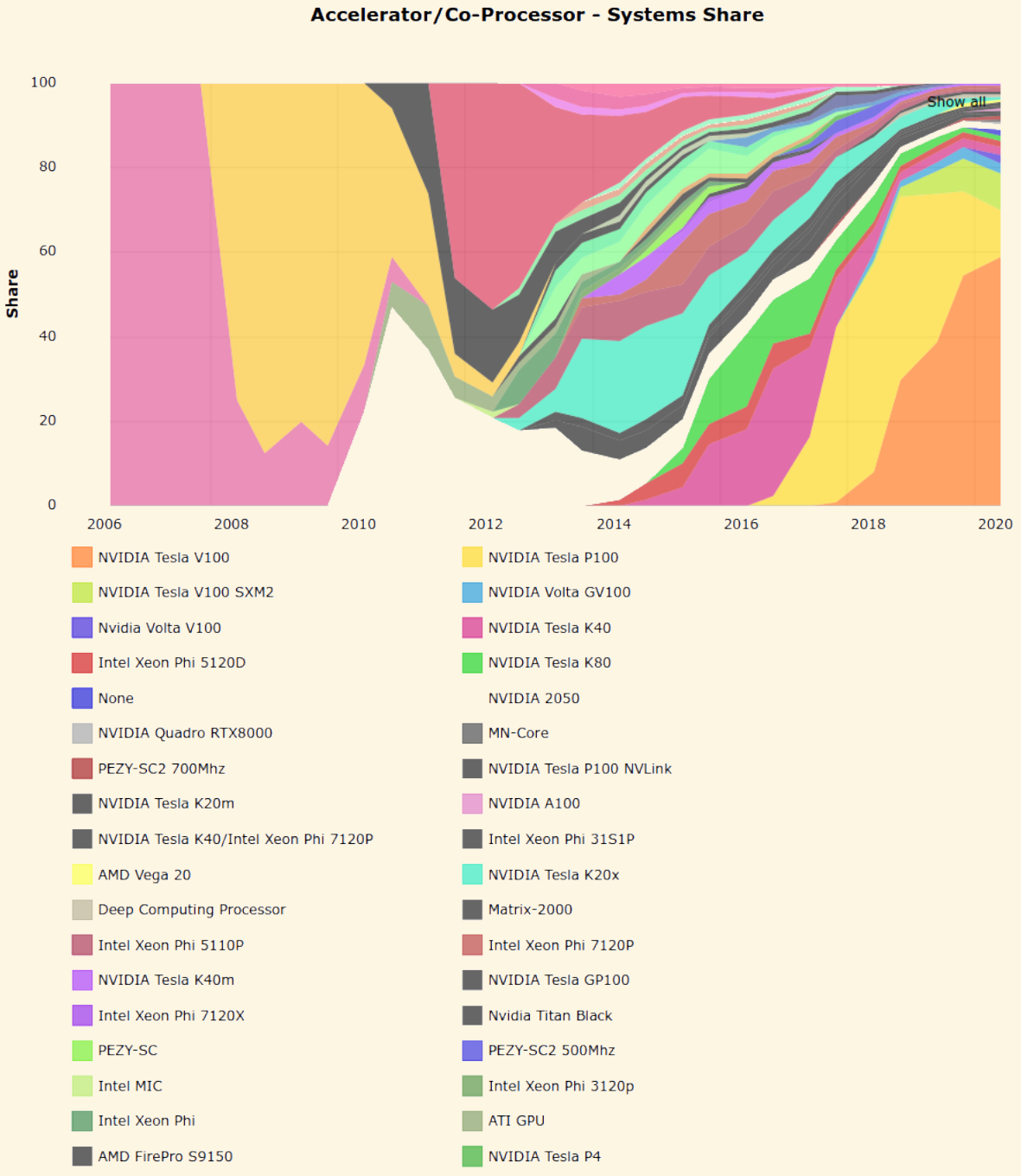
GPU and modern HPC

Site:	**RIKEN Center for Comp. Sci.**	**DOE/SC/Oak Ridge Nat. Lab.**	**DOE/NNSA/LLNL**
Manufacturer:	Fujitsu	IBM	IBM / NVIDIA / Mellanox
Cores:	7,299,072	2,414,592	1,572,480
Memory:	4,866,048 GB	2,801,664 GB	1,382,400 GB
Processor:	A64FX 48C 2.2GHz	IBM POWER9 22C 3.07GHz	IBM POWER9 22C 3.1GHz
Interconnect:	Tofu interconnect D	Dual-rail Infiniband	Dual-rail Infiniband
Performance			
Linpack	415,530 TFlop/s	148,600 TFlop/s	94,640 TFlop/s
Theoretical Peak	513,855 TFlop/s	200,795 TFlop/s	125,712 TFlop/s
Nmax	20,459,520	16,473,600	11,902,464
HPCG [TFlop/s]	13,366.4	2,925.75	1,795.67
Power Consumption			
Power:	28,334.50 kW	10,096.00 kW	7,438.28 kW
Software			
Operating System:	Red Hat Enterprise Linux	RHEL 7.4	RHEL 7.4
Compiler:	FUJITSU Soft. V4.0	XLC, nvcc	IBM XLC
Math Library:	FUJITSU Soft. V4.0	ESSL, CUBLAS 9.2	ESSL, CUBLAS 9.2
MPI:	FUJITSU Soft. V4.0	Spectrum MPI	IBM Spectrum MPI

Table: June 2020: www.top500.org

In order to read about FUGAKU get the report: Jack Dongarra's FUGAKU Report, 22 June 2020

GPU and modern HPC



NVIDIA Supercomputer

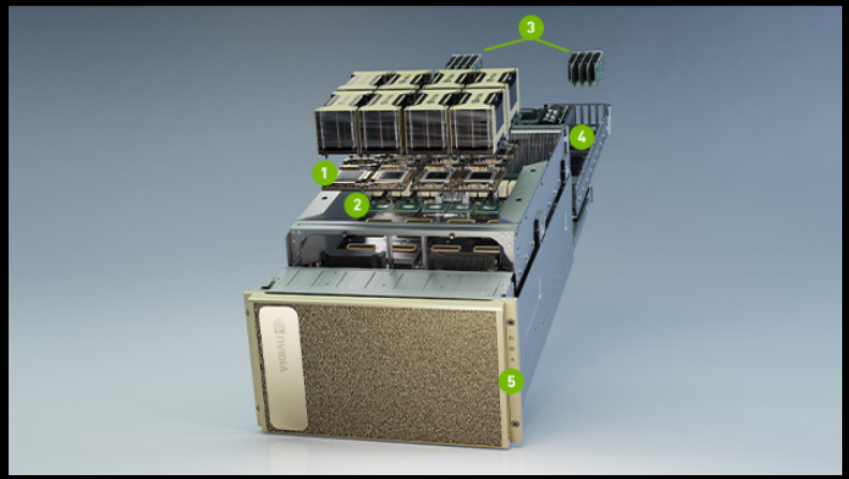
POD Architecture

NVIDIA website

DGX A100 HPC Server

GPU and modern HPC

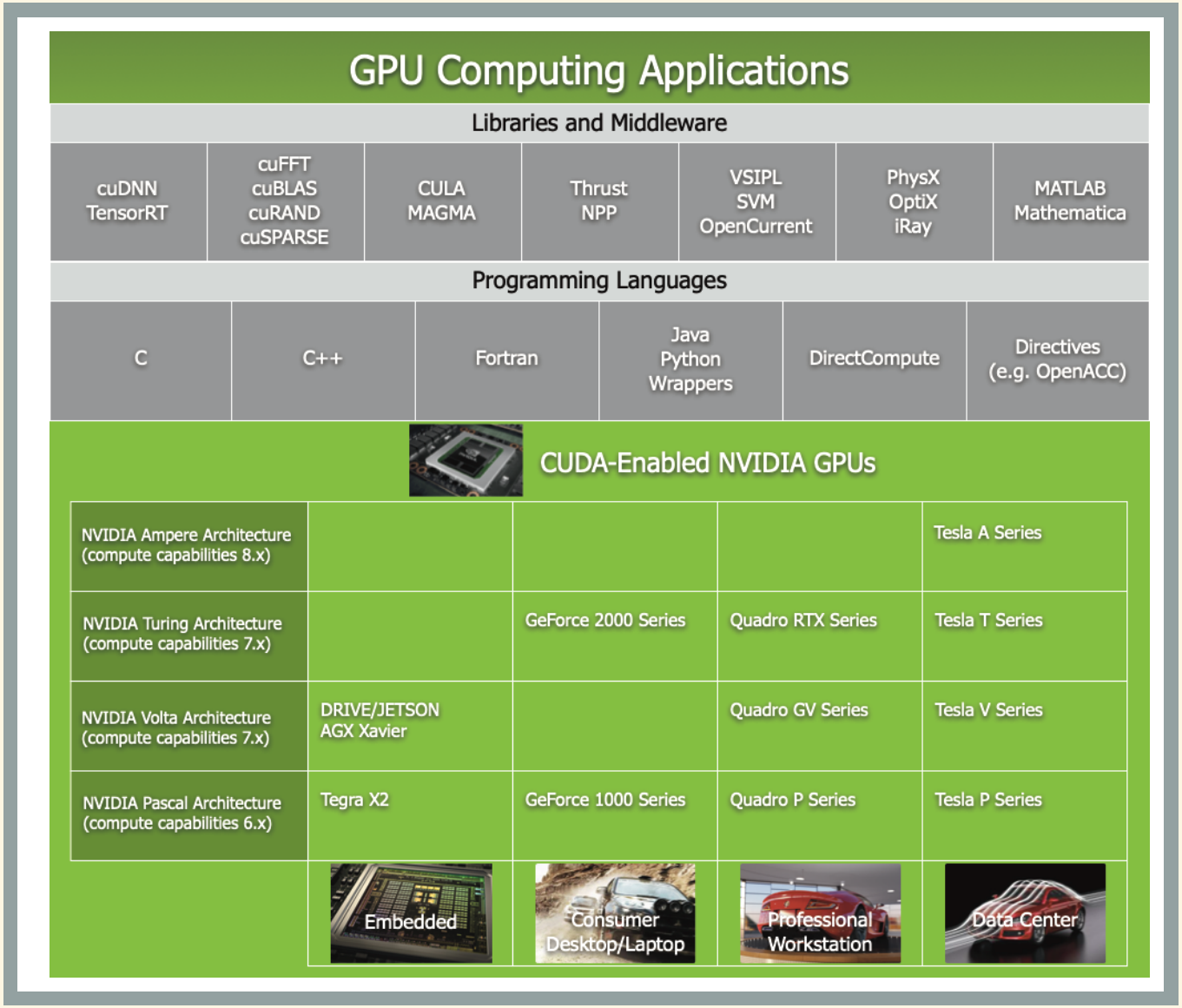
1. 8X NVIDIA A100 GPUS WITH 320 GB TOTAL GPU MEMORY
   12 NVLinks/GPU, 600 GB/s GPU-to-GPU Bi-directonal Bandwidth
2. 6X NVIDIA NVSWITCHES
   4.8 TB/s Bi-directional Bandwidth, 2X More than Previous Generation NVSwitch
3. 9x MELLANOX CONNECTX-6 200Gb/S NETWORK INTERFACE
   450 GB/s Peak Bi-directional Bandwidth
4. DUAL 64-CORE AMD CPUs AND 1 TB SYSTEM MEMORY
   3.2X More Cores to Power the Most Intensive AI Jobs
5. 15 TB GEN4 NVME SSD
   25GB/s Peak Bandwidth, 2X Faster than Gen3 NVME SSDs



NVIDIA website

GPU computing applications

GPU and modern HPC



NVIDIA website

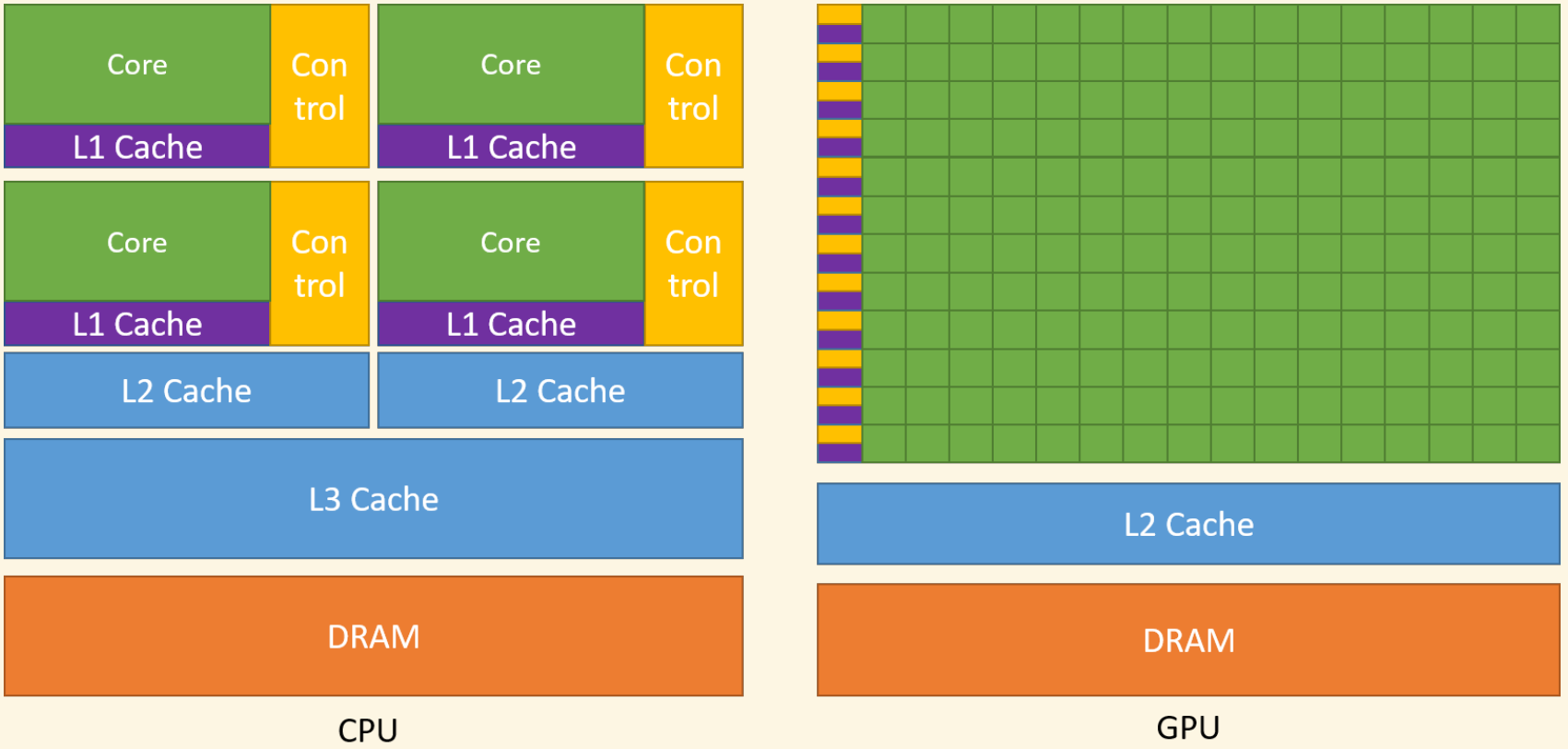
NVIDIA Processors Evolution

GPU and modern HPC

Product Architecture	Pascal P100	Volta V100	NVIDIA A100
Announcement date	April 2016	December 2017	May 2020
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
SMs	56	80	108
TPCs	28	40	54
FP32 Cores / SM	64	64	64
FP32 Cores / GPU	3584	5120	6912
FP64 Cores / SM	32	32	32
FP64 Cores / GPU	1792	2560	3456
INT32 Cores / SM	NA	64	64
INT32 Cores / GPU	NA	5120	6912
Tensor Cores / SM	NA	8	4
Tensor Cores / GPU	NA	640	432
GPU Boost Clock	1480 MHz	1530 MHz	1410 MHz
Peak FP16 TFLOPS1	21.2	31.4	78
Peak FP32 TFLOPS1	10.6	15.7	19.5
Peak FP64 TFLOPS1	5.3	7.8	9.7
Texture Units	224	320	432
Memory Interface	4096-bit HBM2	4096-bit HBM2	5120-bit HBM2
Memory Size	16 GB	32 GB / 16 GB	40 GB
Memory Data Rate	703 MHz DDR	877.5 MHz DDR	1215 MHz DDR
Memory Bandwidth	720 GB/sec	900 GB/sec	1.6 TB/sec
L2 Cache Size	4096 KB	6144 KB	40960 KB
Shared Memory Size / SM	64 KB	up to 96 KB	up to 164 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	14336 KB	20480 KB	27648 KB
TDP	300 Watts	300 Watts	400 Watts
Transistors	15.3 billion	21.1 billion	54.2 billion
GPU Die Size	610 mm^2	815 mm^2	826 mm^2
TSMC Manufact. Proc.	16 nm FinFET+	12 nm FFN	7 nm N7

General Components of a GPU Processor

Introduction to CUDA and GPGPU

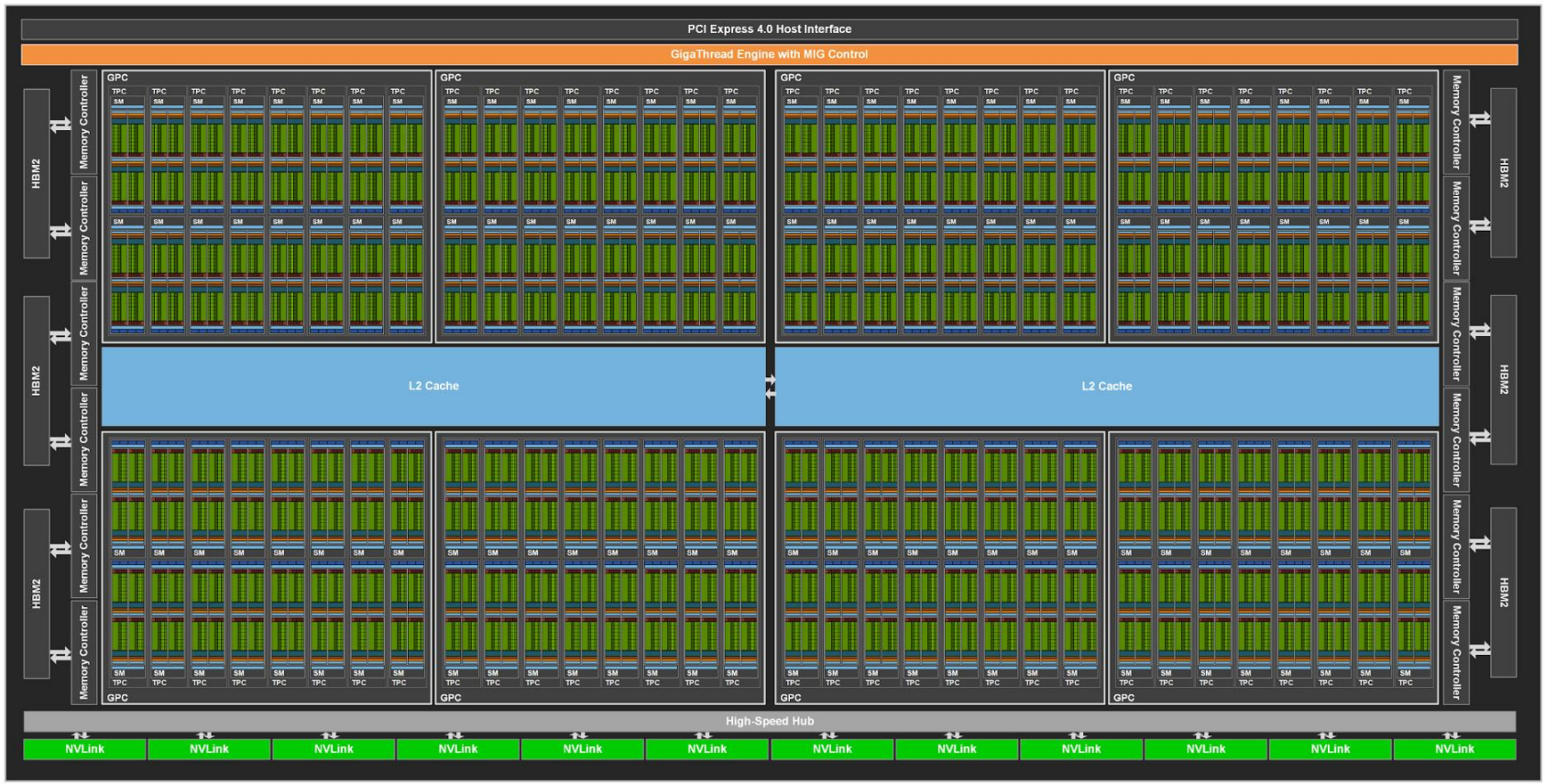


NVIDIA CUDA Programming Guide

Architecture of GA100 Processor

Introduction to CUDA and GPGPU

GA100 Full GPU with 128 SMs (A100 Tensor Core GPU has 108 SMs)



NVIDIA A100 Tensor Core GPU Architecture

GP100 Streaming Multiprocessor Internals

Introduction to CUDA and GPGPU

NVIDIA

Pascal SM consists of:

- 64 (cc 6.0) or 128 (6.1 and 6.2) CUDA cores for arithmetic operations,
- 16 (cc 6.0) or 32 (6.1 and 6.2) special function units for single-precision floating-point,
- 2 (6.0) or 4 (6.1 and 6.2) warp schedulers.

GA100 Streaming Multiprocessor Internals

Introduction to CUDA and GPGPU

Ampere SM consists of:

- 64 FP32 cores for single-precision arithmetic operations,
- 32 FP64 cores for double-precision arithmetic operations,
- 64 INT32 cores for integer math,
- 4 mixed-precision Tensor Cores,
- 16 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

NVIDIA

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction
3. A warp is build of 32 threads

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction
3. A warp is build of 32 threads
4. Warps are gathered in *blocks*

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction
3. A warp is build of 32 threads
4. Warps are gathered in *blocks*
5. One block is assigned to single SM only

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction
3. A warp is build of 32 threads
4. Warps are gathered in *blocks*
5. One block is assigned to single SM only
6. One SM may execute many blocks

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

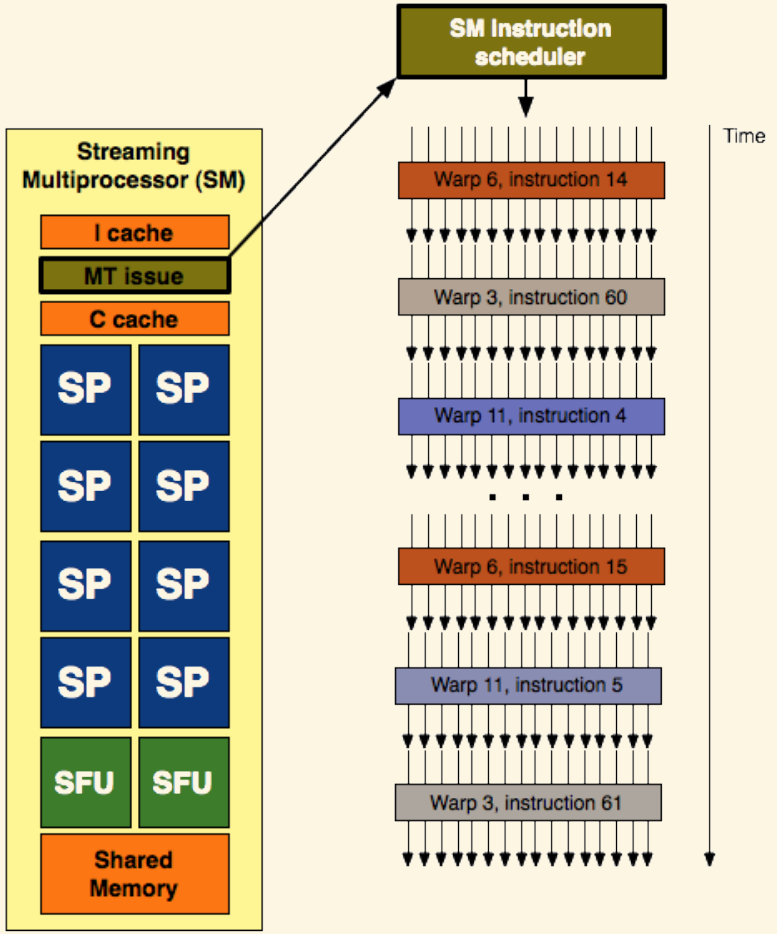
1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction
3. A warp is build of 32 threads
4. Warps are gathered in *blocks*
5. One block is assigned to single SM only
6. One SM may execute many blocks

Threads Execution

Introduction to CUDA and GPGPU

Simplification:

1. Threads are coupled in groups called *warps*
2. Threads in a warp can only perform the same instruction
3. A warp is build of 32 threads
4. Warps are gathered in *blocks*
5. One block is assigned to single SM only
6. One SM may execute many blocks



NVIDIA

Kernels – Threads definitions

Introduction to CUDA and GPGPU

Simplification:

- special C++ function with `__global__` declaration
- compiler runs N CUDA threads in parallel

Kernels – Threads definitions

Introduction to CUDA and GPGPU

Simplification:

- special C++ function with `__global__` declaration
- compiler runs N CUDA threads in parallel

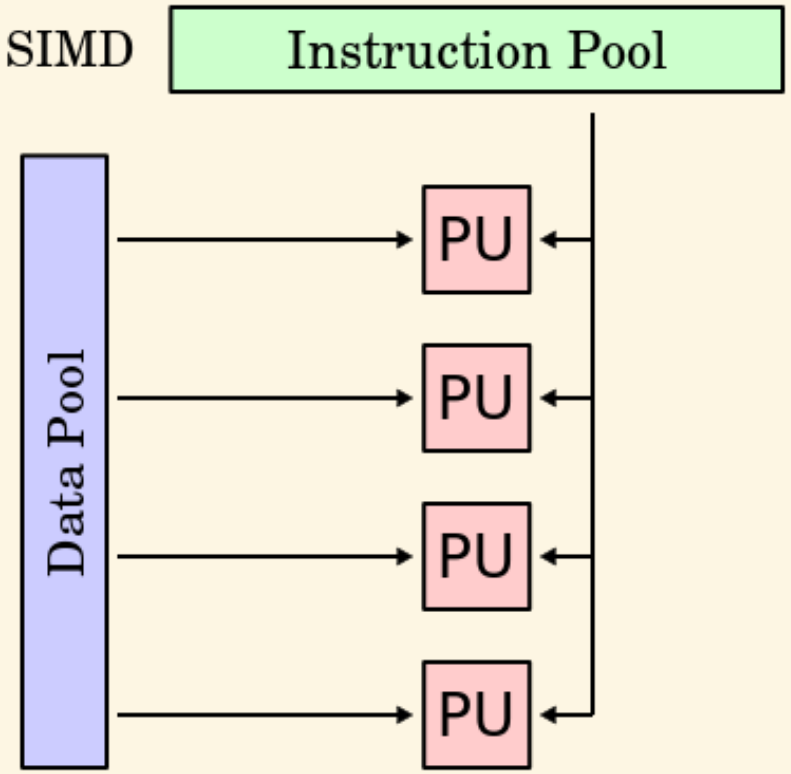
Definition of a kernel:

```
1 __global__ void VecAdd(float* A, float* B, float* C)
2 {
3     int i = threadIdx.x;
4     C[i] = A[i] + B[i];
5 }
```

Invocation of a kernel:

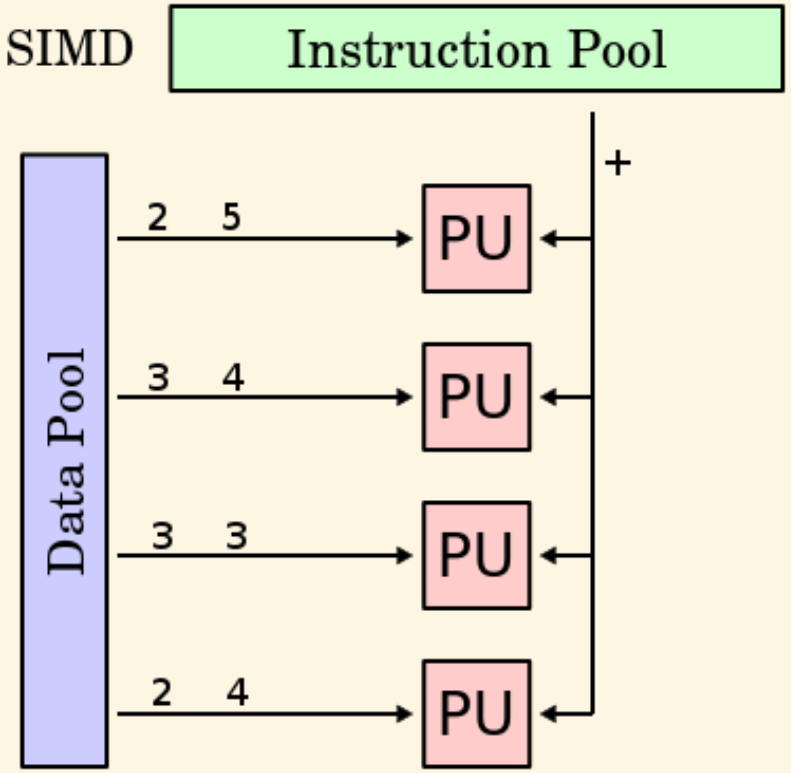
```
1 int main()
2 {
3     VecAdd<<<1, N>>>(A, B, C);
4 }
```

SIMD processing model



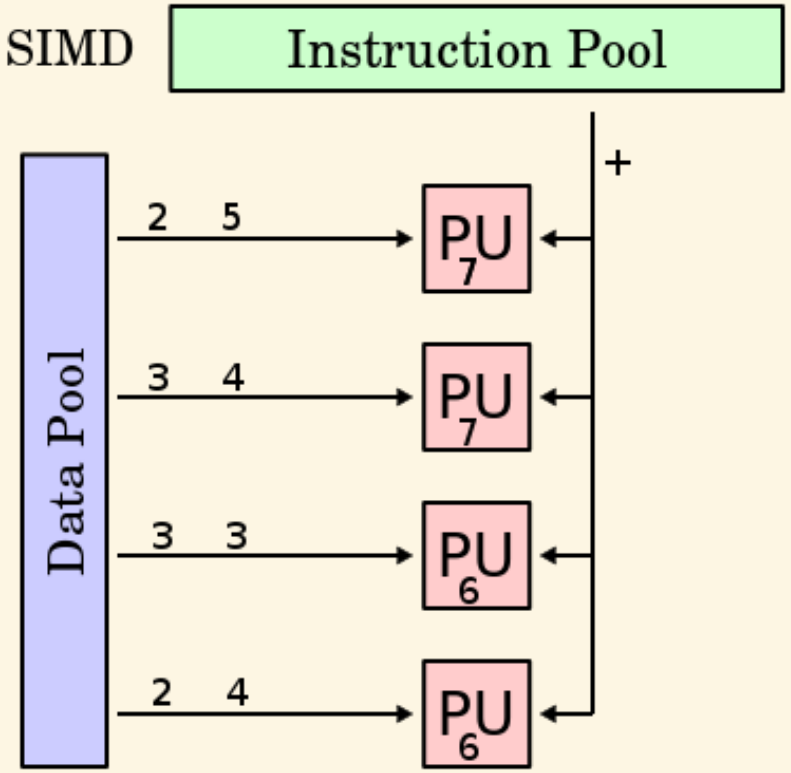
SIMD processing model

Introduction to CUDA and GPGPU



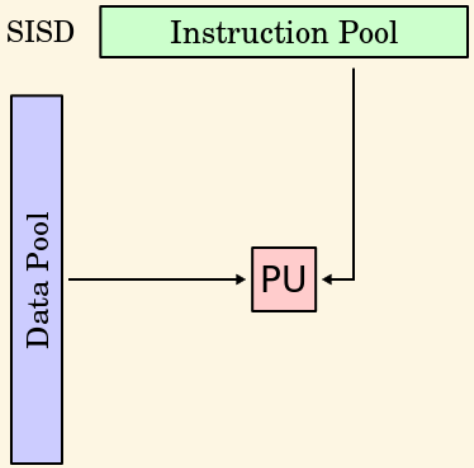
SIMD processing model

Introduction to CUDA and GPGPU



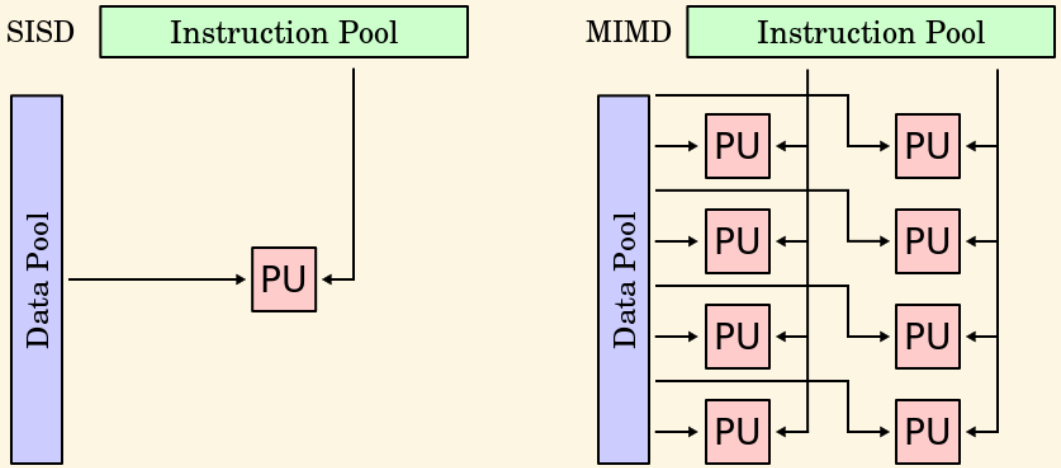
SISD, MIMD, MISD - Flynn Taxonomy

Introduction to CUDA and GPGPU



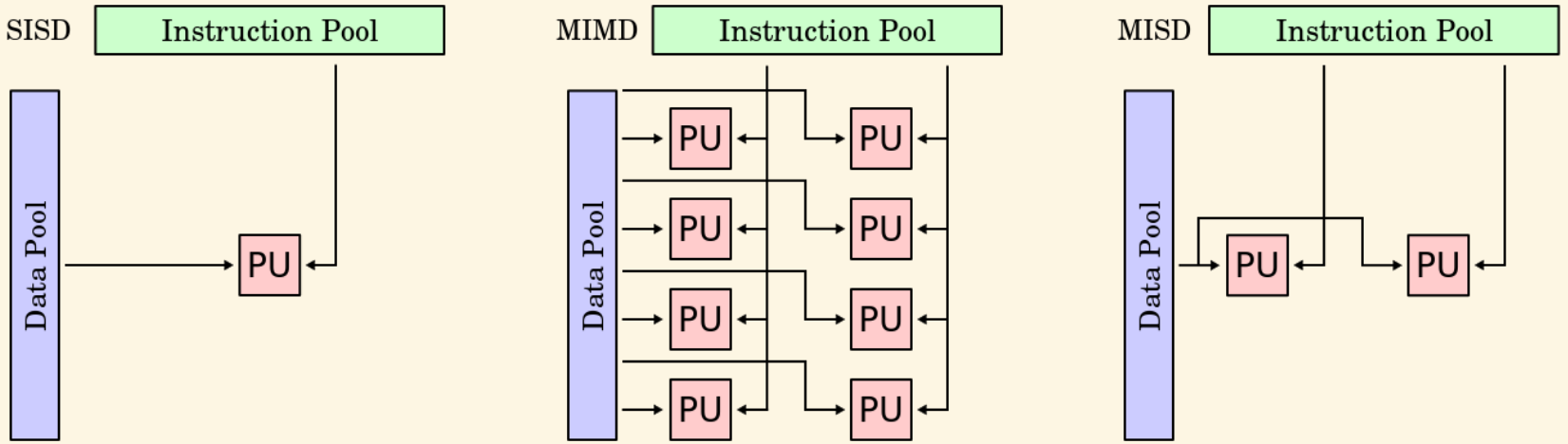
SISD, MIMD, MISD - Flynn Taxonomy

Introduction to CUDA and GPGPU



SISD, MIMD, MISD - Flynn Taxonomy

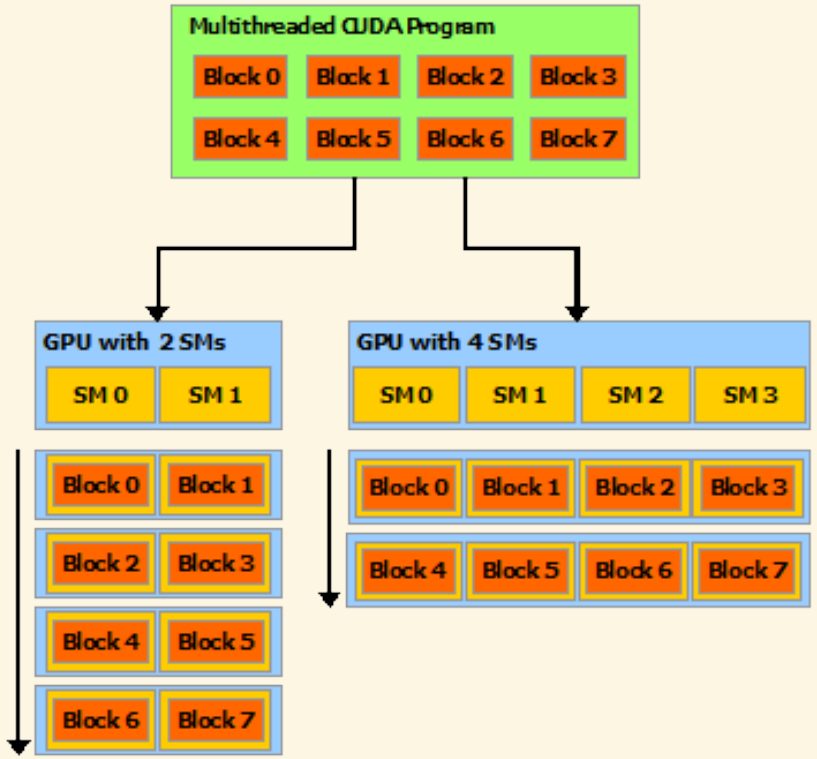
Introduction to CUDA and GPGPU



Automatic threads scalability

Introduction to CUDA and GPGPU

1. Thread blocks are automatically assigned to SMs.

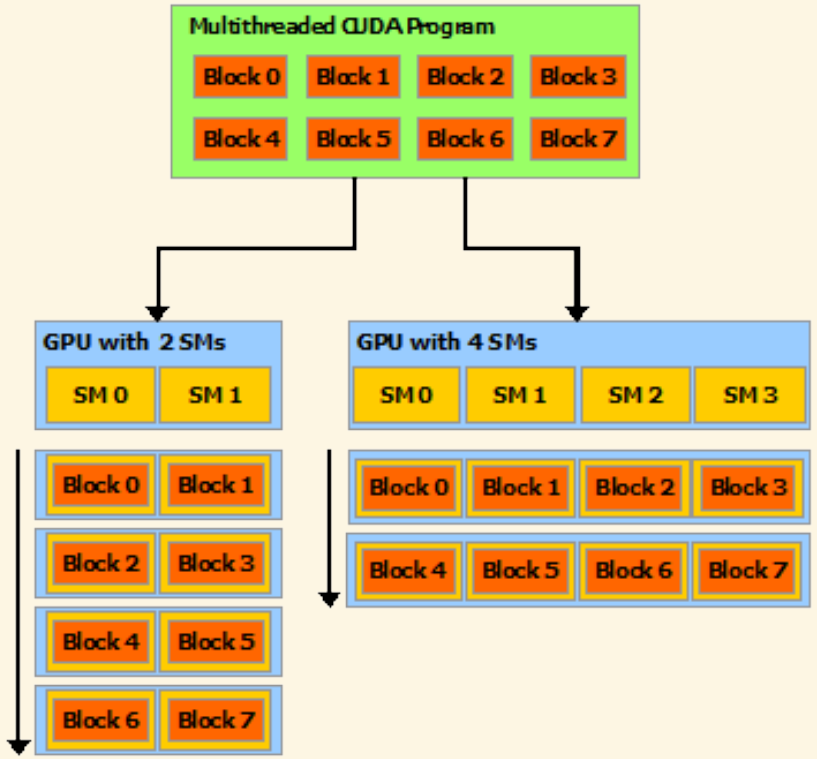


NVIDIA

Automatic threads scalability

Introduction to CUDA and GPGPU

1. Thread blocks are automatically assigned to SMs.
2. Programmers have no control on this process.

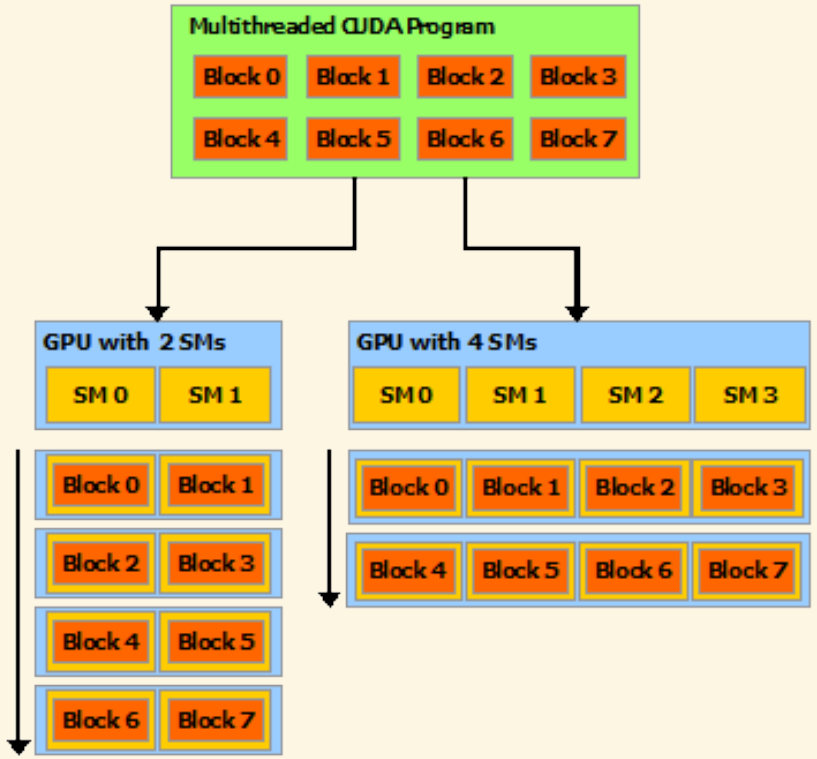


NVIDIA

Automatic threads scalability

Introduction to CUDA and GPGPU

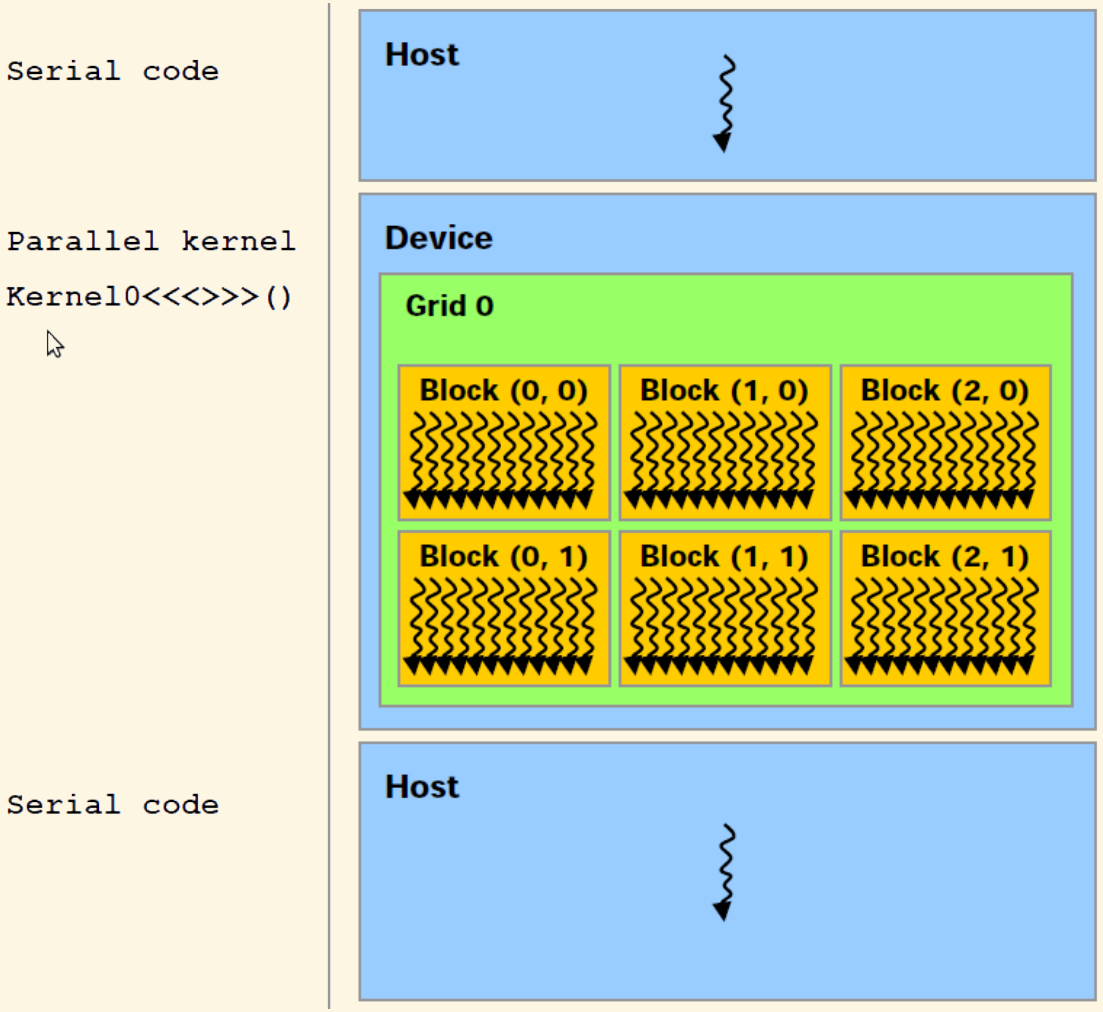
1. Thread blocks are automatically assigned to SMs.
2. Programmers have no control on this process.
3. Subsequent kernel execution may result in different assignment.



NVIDIA

Heterogeneous programming with host and device

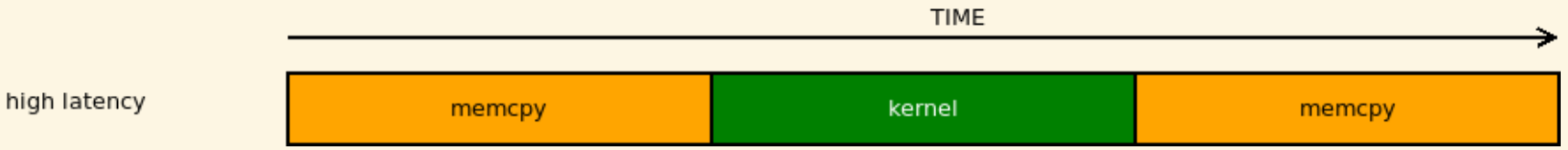
Introduction to CUDA and GPGPU



NVIDIA

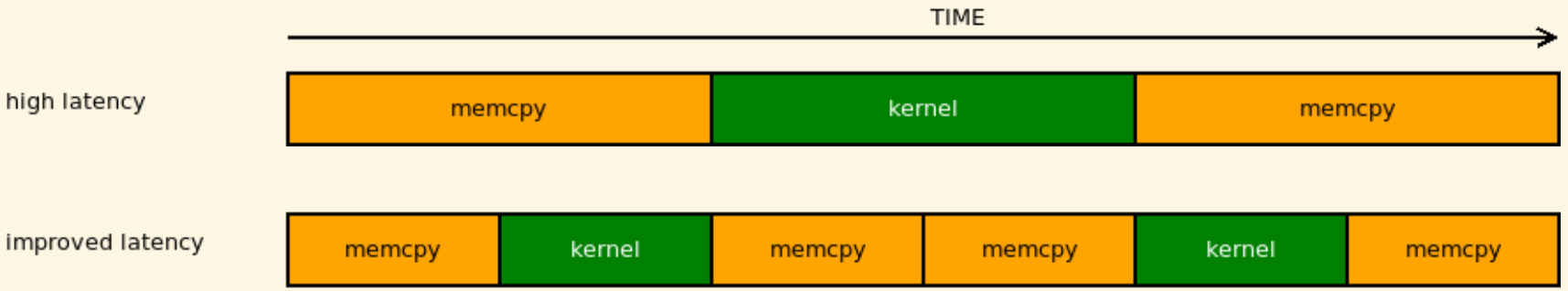
Parallel kernel and memory copying

Introduction to CUDA and GPGPU



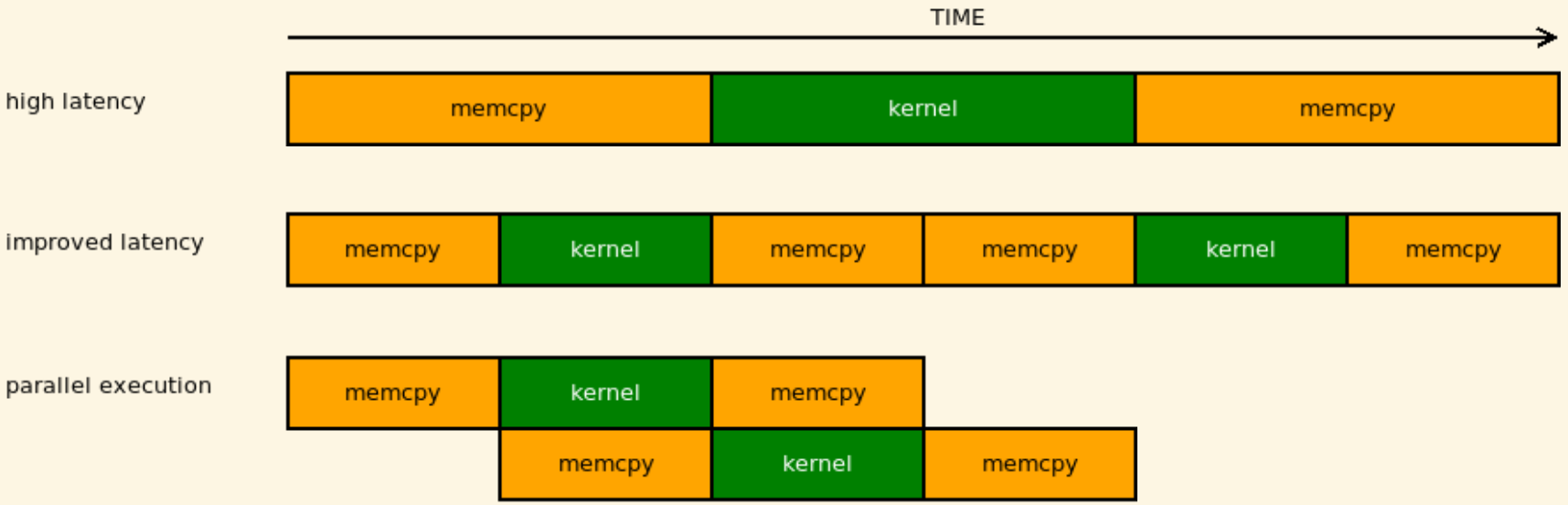
Parallel kernel and memory copying

Introduction to CUDA and GPGPU



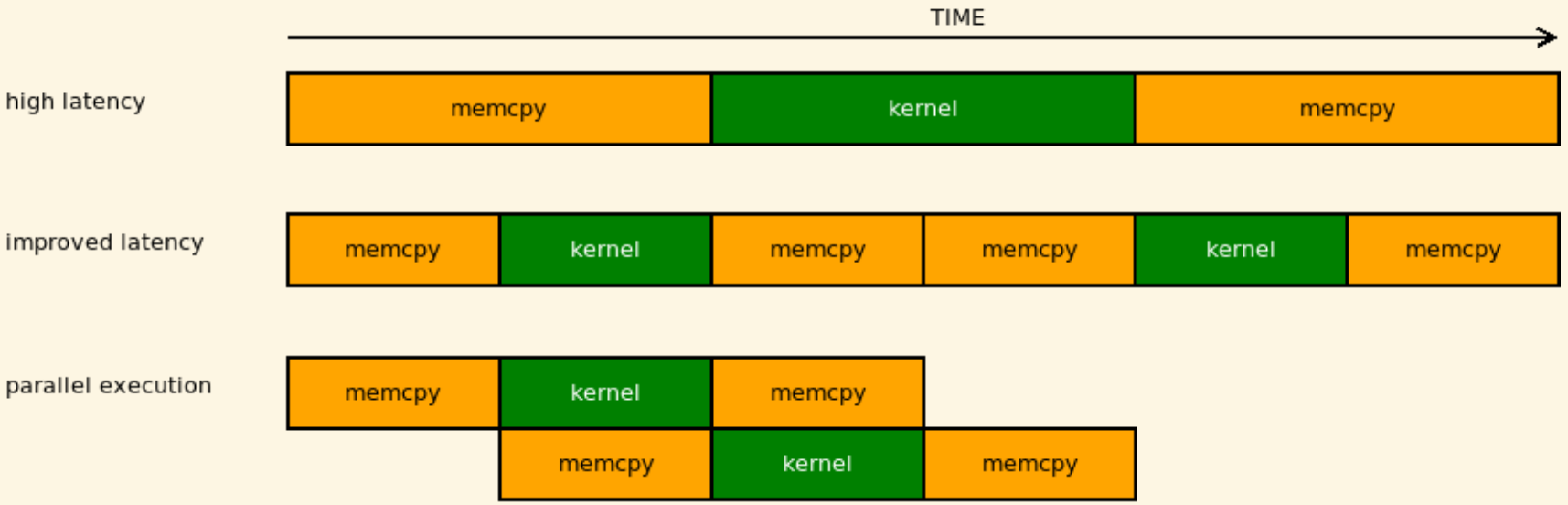
Parallel kernel and memory copying

Introduction to CUDA and GPGPU



Parallel kernel and memory copying

Introduction to CUDA and GPGPU



- Parallel memory copying and kernel execution requires asynchronous (non-blocking) memory copying and execution streams (cuda streams).

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation

CUDA Language Characteristics

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation
  - no variable number of arguments

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation
  - no variable number of arguments
  - no static variables

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation
  - no variable number of arguments
  - no static variables
  - limited recursion

CUDA Language Characteristics

Introduction to CUDA and GPGPU

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation
  - no variable number of arguments
  - no static variables
  - limited recursion
  - must be `void`

CUDA Language Characteristics

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation
  - no variable number of arguments
  - no static variables
  - limited recursion
  - must be `void`
- Kernel launches are **asynchronous** (return to CPU immediately).

CUDA Language Characteristics

- Modified C++ language
- A program is build of C++ functions (executed in CPU or GPU)
- Function running in GPU (streaming processor) is called **kernel**.
- Kernel properties:
  - can only access GPU memory or CPU memory with special allocation
  - no variable number of arguments
  - no static variables
  - limited recursion
  - must be `void`
- Kernel launches are **asynchronous** (return to CPU immediately).
- Kernel executes after all previous CUDA calls have completed.

CUDA Language Characteristics

Threads Identification

- Each kernel contains local variables defining the execution context:

CUDA Language Characteristics

Threads Identification

- Each kernel contains local variables defining the execution context:
  - `threadIdx` – three dimensional value unique within a block

CUDA Language Characteristics

Threads Identification

- Each kernel contains local variables defining the execution context:
  - `threadIdx` – three dimensional value unique within a block
  - `blockIdx` – three dimensional value unique within a grid

CUDA Language Characteristics

Threads Identification

- Each kernel contains local variables defining the execution context:
  - `threadIdx` – three dimensional value unique within a block
  - `blockIdx` – three dimensional value unique within a grid
  - `blockDim` – three dimensional value describing a block dimensions

CUDA Language Characteristics

Threads Identification

- Each kernel contains local variables defining the execution context:
  - `threadIdx` – three dimensional value unique within a block
  - `blockIdx` – three dimensional value unique within a grid
  - `blockDim` – three dimensional value describing a block dimensions
  - `gridDim` – three dimensional value describing a grid dimensions

CUDA Language Characteristics

Defining Grid and Blocks

- Thread block (composed of thread warps) is a group of threads that can:

CUDA Language Characteristics

Defining Grid and Blocks

- Thread block (composed of thread warps) is a group of threads that can:
  - synchronize their execution

CUDA Language Characteristics

Defining Grid and Blocks

- Thread block (composed of thread warps) is a group of threads that can:
  - synchronize their execution
  - communicate via shared memory

CUDA Language Characteristics

Defining Grid and Blocks

- Thread block (composed of thread warps) is a group of threads that can:
  - synchronize their execution
  - communicate via shared memory
- Single block is assigned to a single SM for all its lifetime.

CUDA Language Characteristics

Defining Grid and Blocks

- Thread block (composed of thread warps) is a group of threads that can:
  - synchronize their execution
  - communicate via shared memory
- Single block is assigned to a single SM for all its lifetime.
- Grid = all blocks for given launch

CUDA Language Elements

Introduction to CUDA and GPGPU

Kernel launch syntax:

```
kernel_name<<<gridDim, blockDim, sharedMem, strId>>>(p1,... pN)
```

- `kernel_name` – name of a kernel function with `__global__` declaration

CUDA Language Elements

Introduction to CUDA and GPGPU

Kernel launch syntax:

```
kernel_name<<<gridDim, blockDim, sharedMem, strId>>>(p1,... pN)
```

- `kernel_name` – name of a kernel function with `__global__` declaration
- `gridDim` – dim3 value describing number of blocks in a grid

CUDA Language Elements

Introduction to CUDA and GPGPU

Kernel launch syntax:

```
kernel_name<<<gridDim, blockDim, sharedMem, strId>>>(p1,... pN)
```

- `kernel_name` – name of a kernel function with `__global__` declaration
- `gridDim` – dim3 value describing number of blocks in a grid
- `blockDim` – dim3 value describing number of threads in each block

CUDA Language Elements

Introduction to CUDA and GPGPU

Kernel launch syntax:

```
kernel_name<<<gridDim, blockDim, sharedMem, strId>>>(p1,... pN)
```

- `kernel_name` – name of a kernel function with `__global__` declaration
- `gridDim` – dim3 value describing number of blocks in a grid
- `blockDim` – dim3 value describing number of threads in each block
- `sharedMem` – (optional) size of shared memory allocated for each block in bytes

CUDA Language Elements

Introduction to CUDA and GPGPU

Kernel launch syntax:

```
kernel_name<<<gridDim, blockDim, sharedMem, strId>>>(p1,... pN)
```

- `kernel_name` – name of a kernel function with `__global__` declaration
- `gridDim` – dim3 value describing number of blocks in a grid
- `blockDim` – dim3 value describing number of threads in each block
- `sharedMem` – (optional) size of shared memory allocated for each block in bytes
- `strId` – (optional) identification of a stream for parallel kernel execution (default 0)

CUDA Language Elements

Introduction to CUDA and GPGPU

Kernel launch syntax:

```
kernel_name<<<gridDim, blockDim, sharedMem, strId>>>(p1,... pN)
```

- `kernel_name` – name of a kernel function with `__global__` declaration
- `gridDim` – dim3 value describing number of blocks in a grid
- `blockDim` – dim3 value describing number of threads in each block
- `sharedMem` – (optional) size of shared memory allocated for each block in bytes
- `strId` – (optional) identification of a stream for parallel kernel execution (default 0)
- `p1,... pN` – kernel parameters
  (automatically copied to a device through the constant memory)

CUDA Language Elements

Introduction to CUDA and GPGPU

- `dim3` type:
  - used for indexing and describing blocks of threads and grids
  - can be constructed from one, two and three values
  - based on `uint[3]`, default value: (1,1,1)

CUDA Language Elements

Introduction to CUDA and GPGPU

- `dim3` type:
  - used for indexing and describing blocks of threads and grids
  - can be constructed from one, two and three values
  - based on `uint[3]`, default value: (1,1,1)
- other built-in vector types:
  - [u]{`char`,`short`,`int`,`long`}{1..4}, `float`{1..4}
  - Structures accessed with `x, y, z, w` fields:

```
uint4 param;
int y = param.y;
```

  - They all come with a constructor, for example:

```
int2 make_int2(int x, int y);
```

CUDA Language Elements

Introduction to CUDA and GPGPU

- functions qualifiers:
  - `__global__` launched by CPU on device (must return `void`)
  - `__device__` called from other GPU functions (never CPU)
  - `__host__` can be executed by CPU
    (can be used together with `__device__`)

Two dimensional block execution I

(one block only)

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads

    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Two dimensional block execution II

(more blocks require global threads identification)

```
1  __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
2  {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      int j = blockIdx.y * blockDim.y + threadIdx.y;
5      if (i < N && j < N)
6          C[i][j] = A[i][j] + B[i][j];
7  }
8
9  int main()
10 {
11     ...
12     // Kernel invocation with multiple blocks according to the
           problem size (please note integer division)
13
14     dim3 threadsPerBlock(16, 16);
15     dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
16     MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
17     ...
18 }
```

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`
- `cudaMemcpy(void *dst, void *src, size_t nBytes, enum cudaMemcpyKind direction)`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`
- `cudaMemcpy(void *dst, void *src, size_t nBytes, enum cudaMemcpyKind direction)`
  - `HostToDevice`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`
- `cudaMemcpy(void *dst, void *src, size_t nBytes, enum cudaMemcpyKind direction)`
  - `HostToDevice`
  - `DeviceToHost`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`
- `cudaMemcpy(void *dst, void *src, size_t nBytes, enum cudaMemcpyKind direction)`
  - `HostToDevice`
  - `DeviceToHost`
  - `DeviceToDevice`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`
- `cudaMemcpy(void *dst, void *src, size_t nBytes, enum cudaMemcpyKind direction)`
  - `HostToDevice`
  - `DeviceToHost`
  - `DeviceToDevice`

Allocating and deallocating memory

Classical (manual) approach

```
1 int n = 1024;
2 int nbytes = n*sizeof(int);
3 int *d_array = 0;
```

- `cudaMalloc((void**)&d_array, nbytes)`
- `cudaMemset(d_array, 0, nbytes)`
- `cudaFree(d_array)`
- `cudaMemcpy(void *dst, void *src, size_t nBytes, enum cudaMemcpyKind direction)`
  - `HostToDevice`
  - `DeviceToHost`
  - `DeviceToDevice`

CPU blocking version (also assures that kernels have completed).

Memory Management

Classical (manual) approach

De-referencing normal CPU pointer on GPU will crash (and vice versa).

Good naming practices

- `d_` – device pointers
- `h_` – host pointers
- `s_` – shared memory

Threads Synchronization I

Basics

- Device side: `__syncthreads()`

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- Host side: `cudaDeviceSynchronize()`

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- Host side: `cudaDeviceSynchronize()`
  - Blocks the current CPU thread until all GPU calls are finished.

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- Host side: `cudaDeviceSynchronize()`
  - Blocks the current CPU thread until all GPU calls are finished.
  - Including all streams.

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- Host side: `cudaDeviceSynchronize()`
  - Blocks the current CPU thread until all GPU calls are finished.
  - Including all streams.
  - (formerly `cudaThreadSynchronize()`)

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- Host side: `cudaDeviceSynchronize()`
  - Blocks the current CPU thread until all GPU calls are finished.
  - Including all streams.
  - (formerly `cudaThreadSynchronize()`)

Threads Synchronization I

Basics

- Device side: `__syncthreads()`
  - Synchronizes all threads in a **block**
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid conflicts when accessing shared memory
  - Allowed in conditional code only if the conditional is uniform across the entire thread block
- Host side: `cudaDeviceSynchronize()`
  - Blocks the current CPU thread until all GPU calls are finished.
  - Including all streams.
  - (formerly `cudaThreadSynchronize()`)

Note

There are other more advanced device synchronization methods which will be discussed later

Device Threads Synchronization

Deprecation Warning

`cudaThreadSynchronize()` is now deprecated:

„Note that this function is deprecated because its name does not reflect its behavior. Its functionality is similar to the non-deprecated function `cudaDeviceSynchronize()`, which should be used instead."

NVIDIA. Cuda toolkit documentation v. 11.1.0. https://docs.nvidia.com/cuda/

CUDA Error Check API

- All CUDA calls return error code: `cudaError_t`
  (Except for kernel launches)

CUDA Error Check API

- All CUDA calls return error code: `cudaError_t` (Except for kernel launches)
- `cudaError_t cudaGetLastError(void)`
  – Returns the code for the last error

CUDA Error Check API

Introduction to CUDA and GPGPU

- All CUDA calls return error code: `cudaError_t`
  (Except for kernel launches)
- `cudaError_t cudaGetLastError(void)`
  – Returns the code for the last error
- `char* cudaGetErrorString(cudaError_t code)`
  – Returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError()));
```

CUDA Error Check API

Introduction to CUDA and GPGPU

- All CUDA calls return error code: `cudaError_t`
  (Except for kernel launches)
- `cudaError_t cudaGetLastError(void)`
  – Returns the code for the last error
- `char* cudaGetErrorString(cudaError_t code)`
  – Returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError()));
```

CUDA Error Check API

- All CUDA calls return error code: `cudaError_t`
  (Except for kernel launches)
- `cudaError_t cudaGetLastError(void)`
  – Returns the code for the last error
- `char* cudaGetErrorString(cudaError_t code)`
  – Returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError()));
```

Check for the error only after a kernel has finished executing – kernel calls are asynchronous.

CUDA Debugging

Introduction to CUDA and GPGPU

```
#ifdef DEBUG
   cudaThreadSynchronize();
   cudaError_t error = cudaGetLastError();
   if(error != cudaSuccess)
   {
      printf("CUDA error: %s\n", cudaGetErrorString(error));
      exit(-1);
   }
#endif
```

Compile with: `$ nvcc -DDEBUG program.cu`

First kernel – Host code completed


```
#include<cuda.h>

int main()
{
    cudaSetDevice(cutGetMaxGflopsDeviceId());
    int N = 4096;
    int numBytes = N*N * sizeof(int);
    cudaMalloc((void**)&d_A, numbytes);
    cudaMalloc((void**)&d_B, numbytes);
    cudaMalloc((void**)&d_C, numbytes);

    cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, numBytes, cudaMemcpyHostToDevice);
    cudaMemset(d_C, 0, numBytes);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);

    cudaMemcpy(h_C, d_C, numBytes, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

Bibliography

 NVIDIA. Cuda toolkit documentation v. 11.1.0. https://docs.nvidia.com/cuda/.

Materiały sponsorowane przez:

Projekt „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca” współfinansowany jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Zadanie 10 pn. „Modyfikacja programów studiów na kierunkach prowadzonych przez Wydział Matematyki i Nauk Informacyjnych”, realizowane w ramach projektu „NERW 2 PW. Nauka – Edukacja – Rozwój – Współpraca”, współfinansowanego jest ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Politechnika Warszawska

Unia Europejska
Europejski Fundusz Społeczny