

Graphic Processors in Computational Applications

Lecture 1

Krzysztof Kaczmarski

28 lutego 2017

Outline

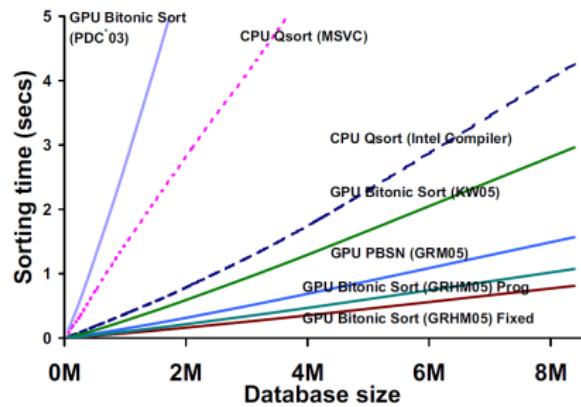
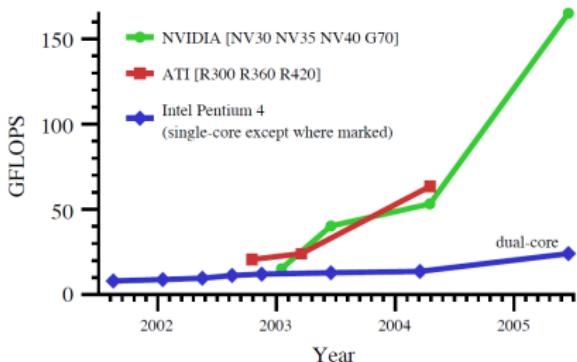
1 Introduction

Parallel Computing's Dark Age

- Impact of data-parallel computing limited
- Thinking Machines (100s of systems total)
- MasPar (sold 200 systems)
- HPC: bring data to supercomputer - no longer possible
- Massively-parallel machines replaced by clusters of ever-more powerful commodity microprocessors
- Beowulf, Legion, grid computing, ...

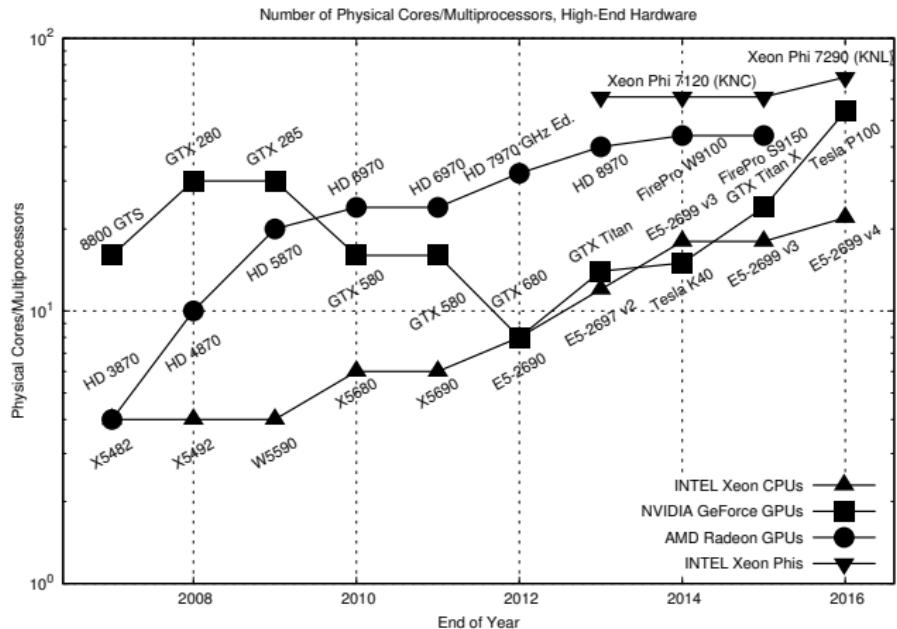
General Purpose Computation on Graphics Processing Unit – GPGPU

Why GPU?

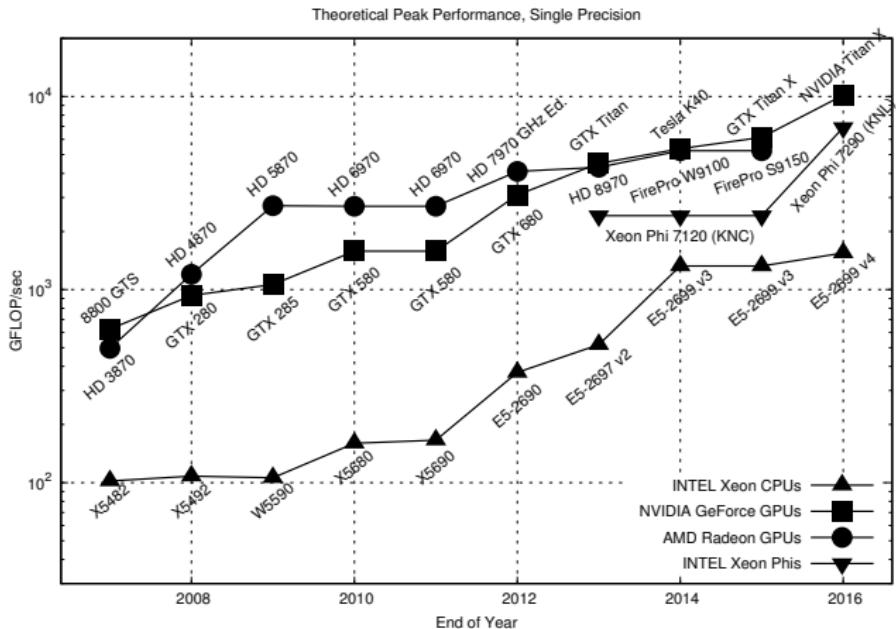


Owens: 2007: ASO

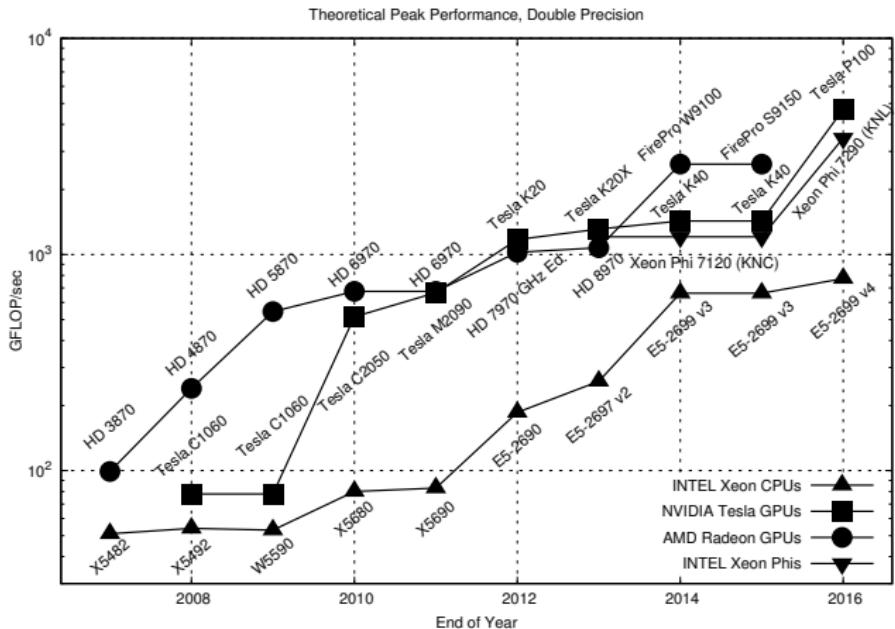
Evolution of Processors



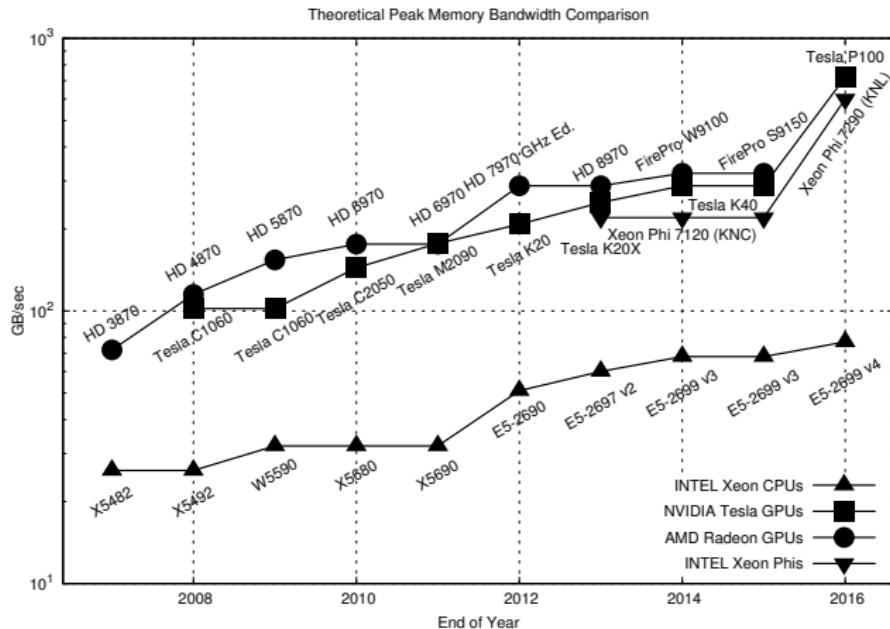
Evolution of Processors



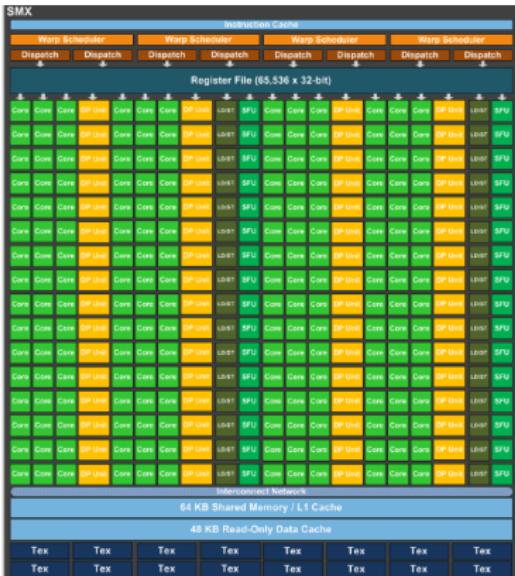
Evolution of Processors



Evolution of Processors



Kepler SM Architecture



SMX: 192 single-precision CUDA cores, 64 double-precision units, 32 special function units [SFU], and 32 load/store units [LD/ST].

Pascal SM Architecture

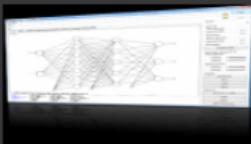


Pascal P100 Device



	TeslaK20M	TeslaK40M	TeslaP100
cc	3.5	3.5	6.0
architecture	kepler	kepler	pascal
SMPs	13	15	56
cores-per-SM	192	192	64
cores	2496	2880	3584
processor	GK110	GK110B	GP100
memory-bandwidth-GBs	208	288	721
threads-per-warp	32	32	32
max-warps-per-SM	64	64	64
memory-bus-bit	320	384	3072
max-threads-per-SM	2048	2048	2048
processor-clock-MHz	706	745	1126
threads-per-block	1024	1024	1024
memory-MB	5120	12288	12288
memory-clock-MHz	1300	1502	704
memory-clock-effective-MHz	5200	6008	1408

GPGPU Applications



GPU Implementation of the Multiple Back-Propagation Algorithm

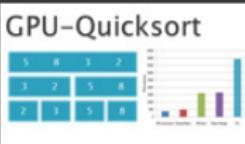
In this paper, we describe a parallel implementation of the Multiple Back-Propagation (MBP) algorithm and present the results obtained when running the algorithm on two well-known benchmarks. The implementation described in the paper will be included in the next version of the Multiple Back-Propagation Software.

Author(s)	Noel Lopes
Organization Type	Academia
Organization	IPG
Software License	

Application Type	Neural Networks
Speed Up	40 x
Date Released	09/01/2009
Available Content	Application / Paper

Embed Code `<div><script`

GPGPU Applications



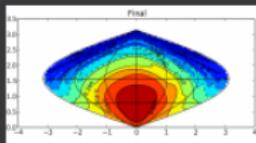
GPU-Quicksort

GPU-Quicksort is a Quicksort-based sorting algorithm designed for GPUs for sorting integers and floats on graphics processors. Experiments shows that it can outperform highly optimized CPU-based Quicksort with a factor of 10 on high-end graphics processor

Author(s)	Daniel Cederman / Philippas Tsigas	Application Type	Libraries
Organization Type	Academia	Speed Up	10 x
Organization	Distributed Computing and Systems - Chalmers University of Technology	Date Released	10/11/2008
Software License	Open source	Available Content	Application / Code / Paper / Presentation

Embed Code `<div><script`

GPGPU Applications



Black holes on GPUs

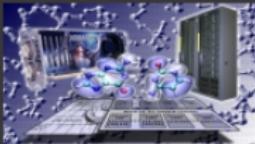
This paper describes a parallel implementation of Monte Carlo simulations using the post-Newtonian equations of motion to model black holes. We use these simulations to investigate the phase space of binary black hole systems.



Author(s)	Frank Herrmann / John Silberholz / Matias ...	Application Type	Numerics / Life Sciences / Science
Organization Type	Academia	Speed Up	50 x
Organization	University of Maryland	Date Released	08/27/2009
Software License		Available Content	Paper

Embed Code `<div><script`

GPGPU Applications



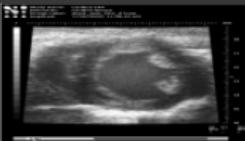
TeraChem

General purpose software for quantum chemistry calculations designed specifically for Nvidia GPU

Author(s)	Ivan Ulfimtsev	Application Type	Life Sciences / Science
Organization Type	Commercial	Speed Up	650 x
Organization	PetaChem, LLC	Date Released	11/24/2009
Software License	Commercial	Available Content	Application / Multimedia

Embed Code `<div><script>`

GPGPU Applications



Heart Wall Tracking

Tracking of mouse heart walls through a series of ultrasound images.

Author(s)	Lukasz G. Szafaryn	Application Type	Medical Imaging
Organization Type	Academia	Speed Up	15 x
Organization	University of Virginia	Date Released	11/05/2009
Software License	Open source	Available Content	Application / Multimedia / Code

Embed Code `<div><script`

GPGPU Applications



CUJ2K - JPEG2000 Encoder

CUJ2K is a fast encoder for the new image compression standard JPEG2000 which is an improvement of JPEG providing better compression ratios and also supporting lossless compression along with many other features. JPEG2000 is very computation-intensive and therefore benefits much from CUDA acceleration. CUJ2K uses streaming to accelerate batch image compression. This program provides commandline-, .Net GUI- and library-interfaces to convert BMP -> JPEG2000. It also supports creation of MJ2 videos.

Author(s)	Norbert Fuerst / Armin Weiss / Simon Papa...	Application Type	Graphics / Imaging / Medical Imaging / Librari..
Organization Type	Hochschule	Speed Up	4 x
Organization	University of Stuttgart, IPVS	Date Released	09/20/2009
Software License	Open Source	Available Content	Application / Paper / Code

Embed Code

GPGPU Applications



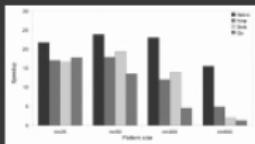
Parallel Algorithm for Solving Kepler's Equation on Graphics Processing Units: Application to Analysis of Doppler Exoplanet Searches

We present the results of a highly parallel Kepler equation solver using the Graphics Processing Unit (GPU) on a commercial nVidia GeForce 280GTX and the "Compute Unified Device Architecture" programming environment. We apply this to evaluate a goodness-of-fit statistic (e.g., χ^2) for Doppler observations of stars potentially harboring multiple planetary companions (assuming negligible planet-planet interactions). We tested multiple implementations using single precision, double precision, pairs of single precision, and mixed precision arithmetic. We find that the vast majority of computations can be performed using

Author(s)	Eric B. Ford	Application Type	Numerics / Science
Organization Type	Academia	Speed Up	600 x
Organization	Department of Astronomy, University of Florida	Date Released	12/16/2009
Software License		Available Content	Paper

Embed Code `<div><script>`

GPGPU Applications



String Matching on a Multicore GPU Using CUDA

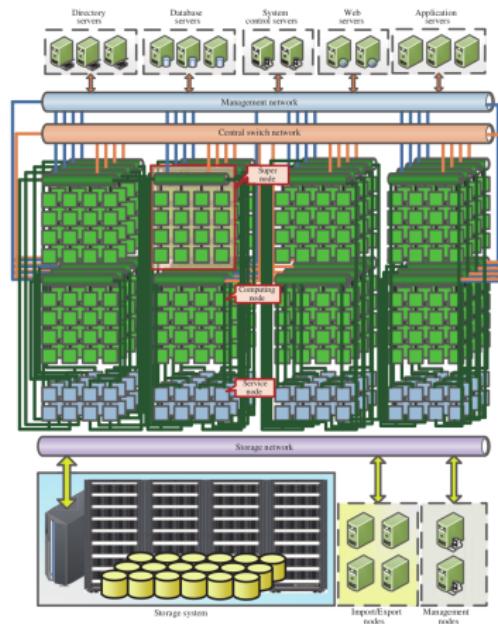
Graphics Processing Units (GPUs) have evolved over the past few years from dedicated graphics rendering devices to powerful parallel processors, outperforming traditional Central Processing Units (CPUs) in many areas of scientific computing. The use of GPUs as processing elements was very limited until recently, when the concept of General-Purpose computing on Graphics Processing Units (GPGPU) was introduced. GPGPU made possible to exploit the processing power and the memory bandwidth of the GPUs with the use of APIs that hide the GPU hardware from programmers. This paper presents experimental results on the parallel processing for some well known on-line string matching algorithms using one such GPU abstraction API, the Compute Unified Device Architecture (CUDA).

Author(s)	C. S. Kouzinopoulos	Application Type	String matching
Organization Type	Academia	Speed Up	24x
Organization	University of Macedonia	Date Released	09/10/2009
Software License		Available Content	Paper

Embed Code `<div><script>`



Sunway TaihuLight



GPGPU Clusters



INTRODUCING TITAN

Advancing the Era of Accelerated Computing

TITAN

U.S. DEPARTMENT OF ENERGY Office of Science

OAK RIDGE National Laboratory

OLCF OAK RIDGE LEADERSHIP COMPUTING FACILITY

www.olcf.ornl.gov/titan/

GPGPU Clusters

Titan will be the first major supercomputing system to utilize a hybrid architecture, or one that utilizes both conventional 16-core AMD Opteron CPUs and NVIDIA Tesla K20 GPU Accelerators.

The combination of CPUs and GPUs will allow Titan and future systems to overcome power and space limitations inherent in previous generations of high-performance computers.

Because they handle hundreds of calculations simultaneously, GPUs can go through many more than CPUs in a given time. Yet they draw only modestly more electricity. By relying on its 299,008 CPU cores to guide simulations and allowing its Tesla K20 GPUs, which are based on NVIDIA's next-generation Kepler architecture to do the heavy lifting, Titan will be approximately ten times more powerful than its predecessor, Jaguar, while occupying the same space and drawing essentially the same level of power.

When complete, Titan will have a theoretical peak performance of more than 20 petaflops, or more than 20,000 trillion calculations per second. This will enable researchers across the scientific arena, from materials to climate change to astrophysics, to acquire unparalleled accuracy in their simulations and achieve research breakthroughs more rapidly than ever before.

TITAN SPECS

PEAK PERFORMANCE
20+
PETAFLOPS

OPTERON CORES
299,008
↑

NVIDIA TESLA
K20 GPU ACCELERATORS
18,688
GPUs

TOTAL SYSTEM MEMORY
710
TERABYTES

COMPUTE NODES
18,688

32GB + 6GB
Memory Per Node
32
6.

GEMINI
INTERCONNECT


4,352 sqft
FLOOR SPACE


GPGPU – The new market...

- NVidia (Tesla, GeForce 8x, Quadro...)
 - Cg (C for graphics), GLSL for OpenGL, HLSL for DirectX
 - PhysX (Ageia - taken by nVidia in February 2008)
 - CUDA (developed from November 2006)
- ATI
 - ATI Stream

GPGPU languages

- CUDA (only NVidia)

GPGPU languages

- CUDA (only NVidia)
 - each next release compatible with previous one
(old software can be run on new hardware)

GPGPU languages

- CUDA (only NVidia)
 - each next release compatible with previous one
(old software can be run on new hardware)
- Stream (only ATI)

GPGPU languages

- CUDA (only NVidia)
 - each next release compatible with previous one
(old software can be run on new hardware)
- Stream (only ATI)
- OpenCL (Open Computing Language) – Kronos Group Consortium

GPGPU languages

- CUDA (only NVidia)
 - each next release compatible with previous one
(old software can be run on new hardware)
- Stream (only ATI)
- OpenCL (Open Computing Language) – Kronos Group Consortium
 - Designed for all GPGPU models and vendors
(currently supported only by NVidia?)

GPGPU languages

- CUDA (only NVidia)
 - each next release compatible with previous one
(old software can be run on new hardware)
- Stream (only ATI)
- OpenCL (Open Computing Language) – Kronos Group Consortium
 - Designed for all GPGPU models and vendors
(currently supported only by NVidia?)
- DirectComputing (for DirectX) – present in Windows 7

GPGPU processing model

- GPU uses SIMD processing model
 - Single Instruction Multiple Data

GPGPU processing model

- GPU uses SIMD processing model
 - Single Instruction Multiple Data
- Threads working in parallel may share common cache

GPGPU processing model

- GPU uses SIMD processing model
 - Single Instruction Multiple Data
- Threads working in parallel may share common cache
- Increased memory data transfer

GPGPU processing model

- GPU uses SIMD processing model
 - Single Instruction Multiple Data
- Threads working in parallel may share common cache
- Increased memory data transfer
- Results:

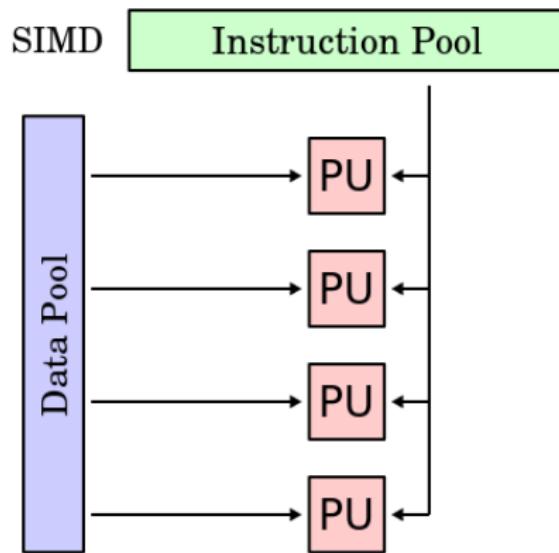
GPGPU processing model

- GPU uses SIMD processing model
 - Single Instruction Multiple Data
- Threads working in parallel may share common cache
- Increased memory data transfer
- Results:
 - Speedup of hundreds times for some applications

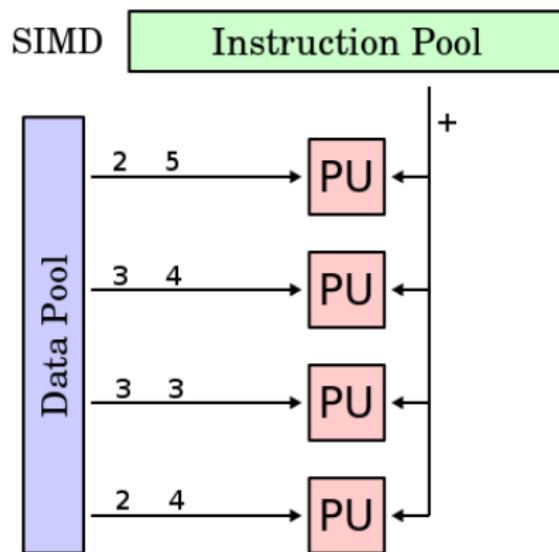
GPGPU processing model

- GPU uses SIMD processing model
 - Single Instruction Multiple Data
- Threads working in parallel may share common cache
- Increased memory data transfer
- Results:
 - Speedup of hundreds times for some applications
 - Extraordinary scalability for different data sets

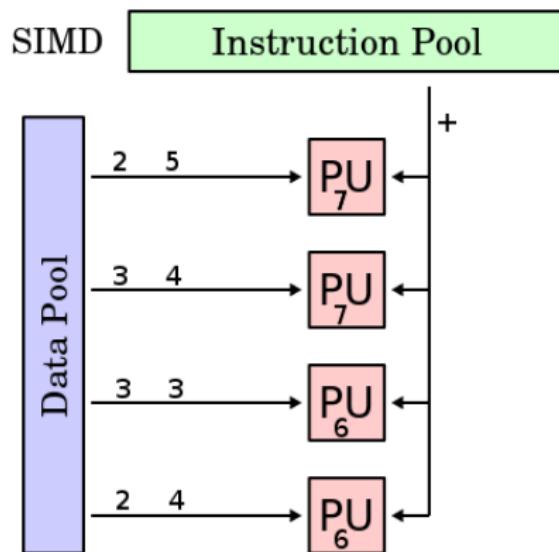
SIMD processing model



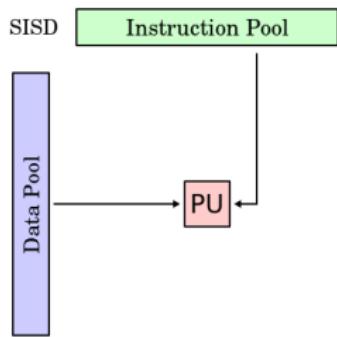
SIMD processing model



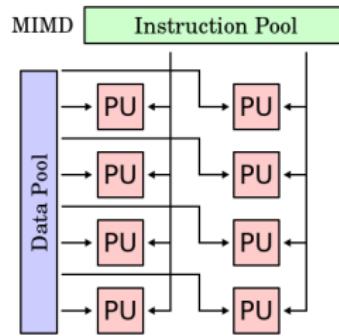
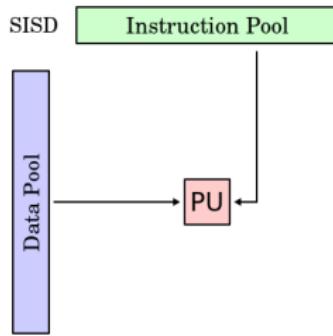
SIMD processing model



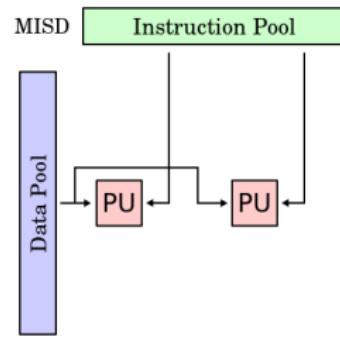
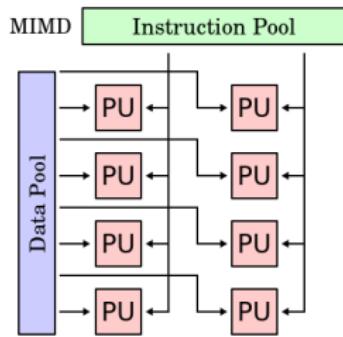
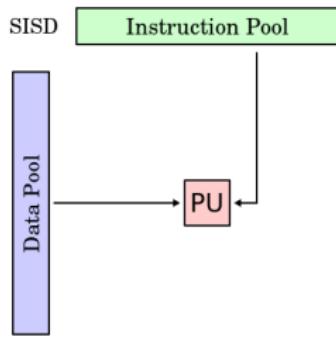
SISD, MIMD, MISD - Flynn Taxonomy



SISD, MIMD, MISD - Flynn Taxonomy



SISD, MIMD, MISD - Flynn Taxonomy

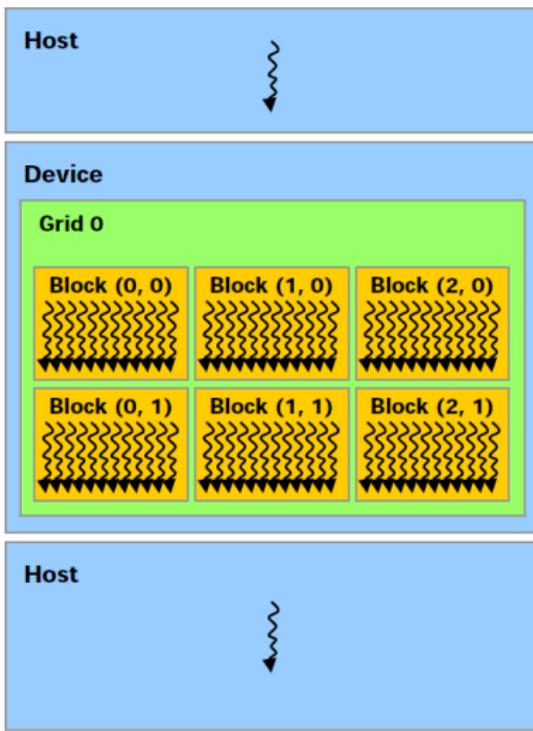


Heterogeneous programming with host and device

Serial code

Parallel kernel

Kernel0<<<>>>()



Serial code

GPGPU drawbacks

- Application must be (re)written especially for GPU

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)
 - However nVidia CUDA seems to be the industry leading solution

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)
 - However nVidia CUDA seems to be the industry leading solution
 - 25,000+ active developers, 100+ applications, 30+ NVIDIA GPU clusters using CUDA tool chain (04/2009)

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)
 - However nVidia CUDA seems to be the industry leading solution
 - 25,000+ active developers, 100+ applications, 30+ NVIDIA GPU clusters using CUDA tool chain (04/2009)
- Task must fit into GPU memory or must be divided into portions

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)
 - However nVidia CUDA seems to be the industry leading solution
 - 25,000+ active developers, 100+ applications, 30+ NVIDIA GPU clusters using CUDA tool chain (04/2009)
- Task must fit into GPU memory or must be divided into portions
- Several programming limitations

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)
 - However nVidia CUDA seems to be the industry leading solution
 - 25,000+ active developers, 100+ applications, 30+ NVIDIA GPU clusters using CUDA tool chain (04/2009)
- Task must fit into GPU memory or must be divided into portions
- Several programming limitations
 - no efficient synchronization mechanisms between threads - huge slow down

GPGPU drawbacks

- Application must be (re)written especially for GPU
- Good scalability for certain tasks (embarrassingly parallel), however not clear algorithms for all tasks
- Dedicated hardware needed
- So far no common standard for programming (currently waiting for DirectX11 and OpenCL)
 - However nVidia CUDA seems to be the industry leading solution
 - 25,000+ active developers, 100+ applications, 30+ NVIDIA GPU clusters using CUDA tool chain (04/2009)
- Task must fit into GPU memory or must be divided into portions
- Several programming limitations
 - no efficient synchronization mechanisms between threads - huge slow down
 - another slow down for computations using conditional constructs

Vector threads execution

All threads in a vector machine perform the same action in the same time.

```
1 if ((thid == num_elements/2) && ((num_elements%2) > 0))
2     temp[num_elements-1] = g_idata[num_elements-1];
3 for (unsigned int i=0; i<num_elements; ++i)
4 {
5     if ( (n+off) <= (num_elements-1) )
6     {
7         if (temp[m+off] > temp[n+off])
8             swap( temp[m+off], temp[n+off] );
9     }
10    off = 1 - off;
11 }
```

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

CUDA characteristics I

- Scales to 100s of cores, 1000s of parallel threads
- Lets programmers truly focus on parallel algorithms only
- Enable heterogeneous systems (i.e., CPU+GPU)

Definitions:

Device = GPU

Host = CPU

Kernel = function called from the host that runs on the device

CUDA APIs:

- A low-level API called the CUDA driver API – `nvcuda` dynamic library (all its entry points prefixed with `cu`).

CUDA APIs:

- A low-level API called the CUDA driver API – `nvcuda` dynamic library (all its entry points prefixed with `cu`).
- A higher-level API called the C run-time for CUDA that is implemented on top of the CUDA driver API – `cudart` dynamic library (all its entry points prefixed with `cuda`).

CUDA APIs:

- A low-level API called the CUDA driver API – `nvcuda` dynamic library (all its entry points prefixed with `cu`).
- A higher-level API called the C run-time for CUDA that is implemented on top of the CUDA driver API – `cudart` dynamic library (all its entry points prefixed with `cuda`).

CUDA APIs:

- A low-level API called the CUDA driver API – `nvcuda` dynamic library (all its entry points prefixed with `cu`).
- A higher-level API called the C run-time for CUDA that is implemented on top of the CUDA driver API – `cudart` dynamic library (all its entry points prefixed with `cuda`).

They are mutually exclusive for older CUDA version. Now in CUDA 4.0 and latest drivers an application may use them together.

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- **Software Development Kit**
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

Compiler

- nvcc – a replacement for standard C compiler (compiler driver)

Compiler

- `nvcc` – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like `cudacc`, `g++`, `cl`, ...)

Compiler

- `nvcc` – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like `cudacc`, `g++`, `cl`, ...)
- `nvcc` can output:

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool

Compiler

- `nvcc` – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like `cudacc`, `g++`, `cl`, ...)
- `nvcc` can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)
 - **-arch**, **-code**, **-gencode** (including **-arch=sm13** for double precision on double enabled devices)

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)
 - **-arch**, **-code**, **-gencode** (including **-arch=sm13** for double precision on double enabled devices)
 - **--ptxas-options=-v** compiler reports total local memory usage per kernel

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)
 - **-arch**, **-code**, **-gencode** (including **-arch=sm13** for double precision on double enabled devices)
 - **--ptxas-options=-v** compiler reports total local memory usage per kernel
 - **--keep** lets intermediate PTX code inspection

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)
 - **-arch**, **-code**, **-gencode** (including **-arch=sm13** for double precision on double enabled devices)
 - **--ptxas-options=-v** compiler reports total local memory usage per kernel
 - **--keep** lets intermediate PTX code inspection
- An executable with CUDA code requires:

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)
 - **-arch**, **-code**, **-gencode** (including **-arch=sm13** for double precision on double enabled devices)
 - **--ptxas-options=-v** compiler reports total local memory usage per kernel
 - **--keep** lets intermediate PTX code inspection
- An executable with CUDA code requires:
 - The CUDA core library (cuda)

Compiler

- **nvcc** – a replacement for standard C compiler (compiler driver)
- Works by invoking all the necessary tools and compilers (like cudacc, g++, cl, ...)
- **nvcc** can output:
 - Either C code (CPU Code) – That must then be compiled with the rest of the application using another tool
 - PTX object code directly
- useful command line parameters:
 - **-deviceemu** (also defines preprocessor macro)
 - **-arch**, **-code**, **-gencode** (including **-arch=sm13** for double precision on double enabled devices)
 - **--ptxas-options=-v** compiler reports total local memory usage per kernel
 - **--keep** lets intermediate PTX code inspection
- An executable with CUDA code requires:
 - The CUDA core library (cuda)
 - The CUDA run-time library (cudart)

Visual Environments

- 1 NVidia NSight for Visual Studio
- 2 NVidia NSight for Linux build on Eclipse
- 3 computeprof – NVidia Profiler

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

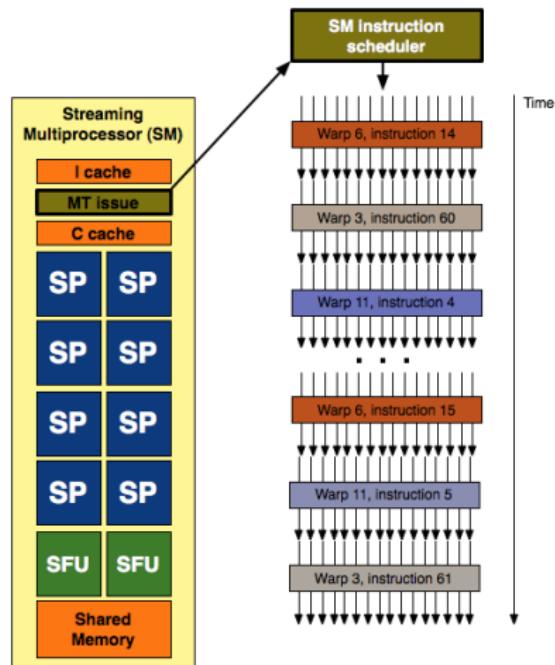
- Introduction
- Software Development Kit
- **A few words about hardware**
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

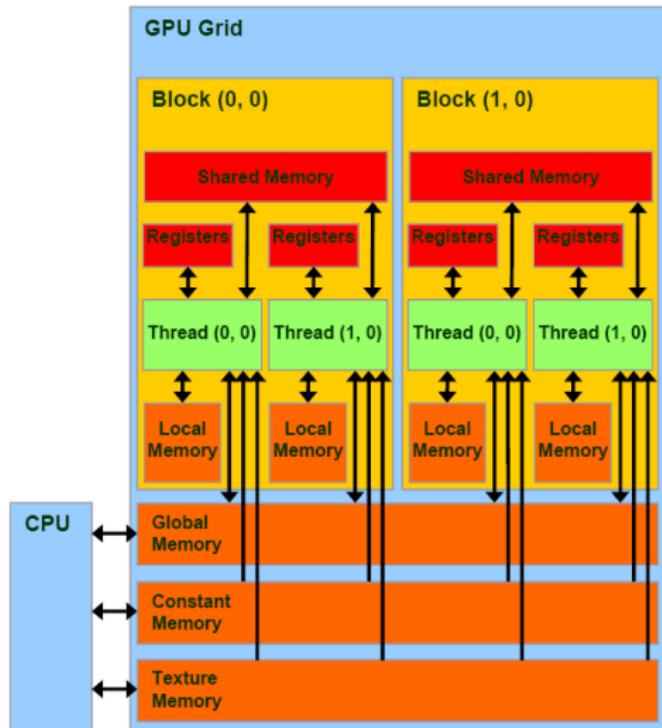
- Volatile

Streaming Multiprocessor

A visualization of a multiprocessor and threads execution (almost accurate)



Memory Hierarchy



Outline

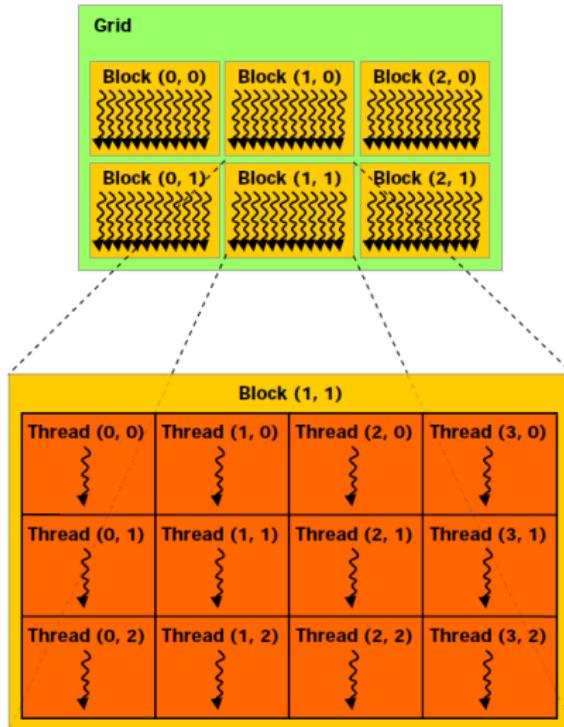
2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution**
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

Threads organization – Grid, Block and Threads



The first kernel

```
1 int blockSize = 32;  
2 dim3 threads( blockSize );  
3 dim3 blocks( arraySize / (float)blockSize + 1 );  
4  
5 first_kernel<<< blocks, threads >>>( arrayPtr, arraySize );
```

The first kernel

```
1 int blockSize = 32;
2 dim3 threads( blockSize );
3 dim3 blocks( arraySize / (float)blockSize + 1 );
4
5 first_kernel<<< blocks, threads >>>( arrayPtr, arraySize );
```

```
1 __global__ void first_kernel(int *a, int dimx)
2 {
3     int ix = blockIdx.x * blockDim.x + threadIdx.x;
4
5     // check if still inside array
6     if (ix < dimx)
7         a[ix] = a[ix] + 1;
8 }
```

The first 2D kernel

```
1 __global__ void first_kernel(int *a, int dimx, int dimy)
2 {
3     int ix = blockIdx.x * blockDim.x + threadIdx.x;
4     int iy = blockIdx.y * blockDim.y + threadIdx.y;
5
6     // check if still inside array
7     if ((ix < dimx) && (iy < dimy))
8     {
9         int idx = iy * dimx + ix;
10        a[idx] = a[idx] + 1;
11        // equivalent: a[ix][iy] = a[ix][iy] + 1;
12    }
13 }
```

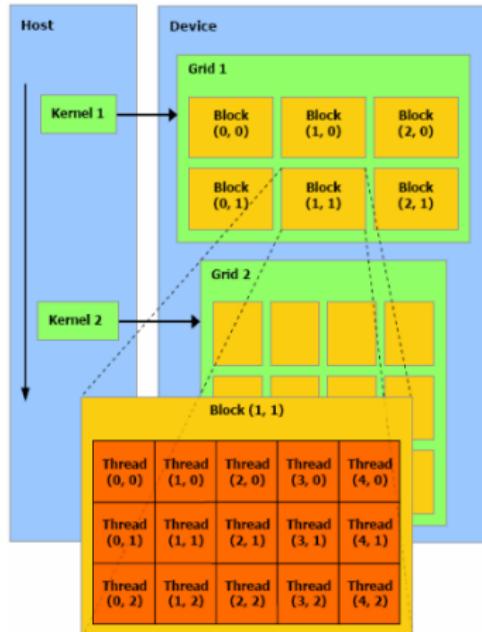
The first 2D kernel

```
1 __global__ void first_kernel(int *a, int dimx, int dimy)
2 {
3     int ix = blockIdx.x * blockDim.x + threadIdx.x;
4     int iy = blockIdx.y * blockDim.y + threadIdx.y;
5
6     // check if still inside array
7     if ((ix < dimx) && (iy < dimy))
8     {
9         int idx = iy * dimx + ix;
10        a[idx] = a[idx] + 1;
11        // equivalent: a[ix][iy] = a[ix][iy] + 1;
12    }
13 }
```

```
1 int blockSize = 32
2 dim3 threads( blockSize, blocksize );
3 dim3 blocks( array_x_dim / (float)blockSize + 1,
4                 array_y_dim / (float)blockSize + 1);
5
6 first_kernel<<< blocks, threads >>>( array_ptr, array_x_dim, array_y_dim );
35 / 111
```

Code must be thread block independent

Blocks may be executed concurrently or sequentially
(depending on context and device capabilities).



CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
 - Threads can cooperatively load data into shared memory

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
 - Threads can cooperatively load data into shared memory
 - Any thread can then access any shared memory location

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
 - Threads can cooperatively load data into shared memory
 - Any thread can then access any shared memory location
- Low learning curve

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
 - Threads can cooperatively load data into shared memory
 - Any thread can then access any shared memory location
- Low learning curve
 - Just a few extensions to C

CUDA Advantages over Legacy GPGPU

- Random access byte-addressable memory
 - Thread can access any memory location
- Unlimited access to memory
 - Thread can read/write as many locations as needed
- Shared memory (per block) and thread synchronization
 - Threads can cooperatively load data into shared memory
 - Any thread can then access any shared memory location
- Low learning curve
 - Just a few extensions to C
 - No knowledge of graphics is required

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

CUDA Language Characteristics I

- Modified C language

CUDA Language Characteristics I

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

CUDA Language Characteristics I

- 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 I

- 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 I

- 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 (now changing in some devices)

CUDA Language Characteristics I

- 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 (now changing in some devices)
 - no variable number of arguments

CUDA Language Characteristics I

- 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 (now changing in some devices)
 - no variable number of arguments
 - no static variables

CUDA Language Characteristics I

- 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 (now changing in some devices)
 - no variable number of arguments
 - no static variables
 - no recursion

CUDA Language Characteristics I

- 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 (now changing in some devices)
 - no variable number of arguments
 - no static variables
 - no recursion
 - must be **void**

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.
- Each thread is free to execute a unique code path
(however with danger of a huge slowdown).

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.
- Each thread is free to execute a unique code path (however with danger of a huge slowdown).
- Each kernel contains local variables defining the execution context:

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.
- Each thread is free to execute a unique code path (however with danger of a huge slowdown).
- Each kernel contains local variables defining the execution context:
 - `threadIdx` – three dimensional value unique within a block

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.
- Each thread is free to execute a unique code path (however with danger of a huge slowdown).
- Each kernel contains local variables defining the execution context:
 - `threadIdx` – three dimensional value unique within a block
 - `blockIdx` – two dimensional value unique within a grid

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.
- Each thread is free to execute a unique code path (however with danger of a huge slowdown).
- Each kernel contains local variables defining the execution context:
 - `threadIdx` – three dimensional value unique within a block
 - `blockIdx` – two dimensional value unique within a grid
 - `blockDim` – two dimensional value describing a block dimensions

CUDA Language Characteristics II – Threads

- We write a kernel from a single thread's point of view.
- Each thread is free to execute a unique code path (however with danger of a huge slowdown).
- Each kernel contains local variables defining the execution context:
 - `threadIdx` – three dimensional value unique within a block
 - `blockIdx` – two dimensional value unique within a grid
 - `blockDim` – two dimensional value describing a block dimensions
 - `gridDim` – two dimensional value describing a grid dimensions

CUDA Language Characteristics III – Blocks

- Thread block is a group of threads that can:

CUDA Language Characteristics III – Blocks

- Thread block is a group of threads that can:
 - synchronize their execution

CUDA Language Characteristics III – Blocks

- Thread block is a group of threads that can:
 - synchronize their execution
 - communicate via shared memory

CUDA Language Characteristics III – Blocks

- Thread block is a group of threads that can:
 - synchronize their execution
 - communicate via shared memory
- Grid = all blocks for given launch

CUDA Language Elements

- **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

- **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:

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

CUDA Language Elements

- 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__`)

CUDA Language Elements

- kernel launch:

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

CUDA Language Elements

- kernel launch:

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

- `f_name` – name of a kernel function with `__global__` qualifier

CUDA Language Elements

- kernel launch:

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

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

CUDA Language Elements

- kernel launch:

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

- **f_name** – name of a kernel function with **__global__** qualifier
- **gridDim** – dim3 value describing number of blocks in a grid
- **blockDim** – dim3 value describing number of threads in each block

CUDA Language Elements

- kernel launch:

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

- **f_name** – name of a kernel function with **__global__** qualifier
- **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 (max 16kB)

CUDA Language Elements

- kernel launch:

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

- **f_name** – name of a kernel function with **__global__** qualifier
- **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 (max 16kB)
- **strId** – (optional) identification of a stream for parallel kernel execution

CUDA Language Elements

- kernel launch:

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

- `f_name` – name of a kernel function with `__global__` qualifier
- `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 (max 16kB)
- `strId` – (optional) identification of a stream for parallel kernel execution
- `p1, ... pN` – kernel parameters (automatically copied to device)

CUDA Language Elements

- kernel launch:

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

- `f_name` – name of a kernel function with `__global__` qualifier
- `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 (max 16kB)
- `strId` – (optional) identification of a stream for parallel kernel execution
- `p1, ... pN` – kernel parameters (automatically copied to device)

- Kernel launches are **asynchronous** (returns to CPU immediately).

CUDA Language Elements

- kernel launch:

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

- `f_name` – name of a kernel function with `__global__` qualifier
- `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 (max 16kB)
- `strId` – (optional) identification of a stream for parallel kernel execution
- `p1, ... pN` – kernel parameters (automatically copied to device)

- Kernel launches are **asynchronous** (returns to CPU immediately).
- Kernel executes after all previous CUDA calls have completed.

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- **Memory Management**
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

Allocating and deallocating memory

Allocating and deallocating memory

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

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

Allocating and deallocating memory

```
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

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

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

Allocating and deallocating memory

```
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

```
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

```
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

```
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

```
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

```
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 has completed).

Memory Management

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

Good naming practices

`d_` – device pointers

`h_` – host pointers

`s_` – shared memory

First kernel – Host code completed

```
1 cudaSetDevice( cutGetMaxGflopsDeviceId() );
2
3 int arraySize = ?;
4 int blockSize = 32;
5 dim3 threads( blockSize );
6 dim3 blocks( arraySize / (float)blockSize );
7 int numBytes = arraySize * sizeof(int);
8 int* h_A = (int*) malloc(numBytes);
9
10 int* d_A = 0;
11 cudaMalloc((void**)&d_A, numbytes);
12
13 cudaMemcpy(d_A, h_A, numBytes, cudaMemcpyHostToDevice);
14
15 first_kernel<<< blocks, threads >>>( d_A, arraySize );
16
17 cudaMemcpy(h_A, d_A, numBytes, cudaMemcpyDeviceToHost);
18
19 cudaFree(d_A);
20 free(h_A);
```

Variable Qualifiers (GPU side)

- `--device--`

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (**`--device--`** qualifier implied)

Variable Qualifiers (GPU side)

- **`__device__`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`__device__` qualifier implied)
 - Accessible by all threads

Variable Qualifiers (GPU side)

- **`__device__`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`__device__` qualifier implied)
 - Accessible by all threads
 - Lifetime: application

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with cudaMalloc (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application

- **`--shared--`**

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with cudaMalloc (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application

- **`--shared--`**

- Stored in on-chip shared memory (very low latency)

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application

- **`--shared--`**

- Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or declared at compile time

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application

- **`--shared--`**

- Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or declared at compile time
 - Accessible by all threads in the same thread block

Variable Qualifiers (GPU side)

- **`--device--`**

- Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application

- **`--shared--`**

- Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or declared at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: kernel execution

Variable Qualifiers (GPU side)

- **`--device--`**
 - Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application
- **`--shared--`**
 - Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or declared at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: kernel execution
- Unqualified variables:

Variable Qualifiers (GPU side)

- **`--device--`**
 - Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application
- **`--shared--`**
 - Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or declared at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: kernel execution
- Unqualified variables:
 - Scalars and built-in vector types are stored in registers

Variable Qualifiers (GPU side)

- **`--device--`**
 - Stored in device memory (large, high latency, no cache)
 - Allocated with `cudaMalloc` (`--device--` qualifier implied)
 - Accessible by all threads
 - Lifetime: application
- **`--shared--`**
 - Stored in on-chip shared memory (very low latency)
 - Allocated by execution configuration or declared at compile time
 - Accessible by all threads in the same thread block
 - Lifetime: kernel execution
- Unqualified variables:
 - Scalars and built-in vector types are stored in registers
 - Arrays of more than 4 elements stored in device memory

Allocating shared memory

■ Device side:

```
1 ^__global__ void kernel(...)  
2 __{  
3   ...  
4   __shared__ float sData[256];  
5   ...  
6 }  
7 __
```

■ Host side:

```
1 __kernel<<<nBlocks, blockSize  
      >>>(...);  
2 __
```

■ Device side:

```
1 __global__ void kernel(...)  
2 __{  
3   ...  
4   extern __shared__ float sData  
     __;  
5   ...  
6 }  
7 __
```

■ Host side:

```
1 __smBytes = blockSize*sizeof(float);  
2  
3 __kernel<<<nBlocks, blockSize,  
4 __           smBytes>>>(...);  
5 __
```

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- **Synchronization**
- Error reporting

3 Advanced topics

- Volatile

Threads Synchronization

- Device side: `__syncthreads()`

Threads Synchronization

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

Threads Synchronization

- 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

- 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

- 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

- 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: `cudaThreadSynchronize()`

Threads Synchronization

- 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: `cudaThreadSynchronize()`
 - Blocks the current CPU thread until all GPU calls are finished.

Threads Synchronization

- 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: `cudaThreadSynchronize()`
 - Blocks the current CPU thread until all GPU calls are finished.
 - Including all streams.

Advanced Threads Synchronization I

CUDA Toolkit > 3.0 and CC \geq 2.0

Device side:

- `int __syncthreads_count(int predicate);` is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.

Advanced Threads Synchronization I

CUDA Toolkit > 3.0 and CC \geq 2.0

Device side:

- `int __syncthreads_count(int predicate);` is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.
- `int __syncthreads_and(int predicate);` similarly but evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for all of them.

Advanced Threads Synchronization I

CUDA Toolkit > 3.0 and CC \geq 2.0

Device side:

- `int __syncthreads_count(int predicate);` is identical to `__syncthreads()` with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.
- `int __syncthreads_and(int predicate);` similarly but evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for all of them.
- `int __syncthreads_or(int predicate);` ... similarly but returns non-zero if predicate evaluates to non-zero for any of the threads.

Advanced Threads Synchronization II

Device side memory fence functions:

- `void __threadfence_block();` waits until all global and shared memory accesses made by the calling thread before are visible to all threads in the thread block.

Advanced Threads Synchronization II

Device side memory fence functions:

- `void __threadfence_block();` waits until all global and shared memory accesses made by the calling thread before are visible to all threads in the thread block.
- `void __threadfence();` waits until all global and shared memory accesses made by the calling thread prior to `__threadfence()` are visible to:

Advanced Threads Synchronization II

Device side memory fence functions:

- `void __threadfence_block();` waits until all global and shared memory accesses made by the calling thread before are visible to all threads in the thread block.
- `void __threadfence();` waits until all global and shared memory accesses made by the calling thread prior to `__threadfence()` are visible to:
 - All threads in the thread block for shared memory accesses,

Advanced Threads Synchronization II

Device side memory fence functions:

- `void __threadfence_block();` waits until all global and shared memory accesses made by the calling thread before are visible to all threads in the thread block.
- `void __threadfence();` waits until all global and shared memory accesses made by the calling thread prior to `__threadfence()` are visible to:
 - All threads in the thread block for shared memory accesses,
 - All threads in the device for global memory accesses.

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

CUDA Error Reporting to CPU

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

CUDA Error Reporting to CPU

- 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 Reporting to CPU

- All CUDA calls return error code: `cudaError_t` (Except for kernel launches)
- `cudaError_t cudaGetLastError(void)` – Returns the code for the last error
- `char* cudaGetString(cudaError_t code)` – Returns a null-terminated character string describing the error
`printf("%s\n", cudaGetString(cudaGetLastError()));`

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

Outline

2 NVidia Compute Unified Device Architecture (CUDA)

- Introduction
- Software Development Kit
- A few words about hardware
- A simple kernel execution
- CUDA Programming Language
- Memory Management
- Synchronization
- Error reporting

3 Advanced topics

- Volatile

Volatile variables and specific compiler optimizations

- One of the compiler's tricks: reuse references to memory location

Volatile variables and specific compiler optimizations

- One of the compiler's tricks: reuse references to memory location
- Result: A reused value may be changed by another thread in the background

Volatile variables and specific compiler optimizations

- One of the compiler's tricks: reuse references to memory location
- Result: A reused value may be changed by another thread in the background

Volatile variables and specific compiler optimizations

- One of the compiler's tricks: reuse references to memory location
- Result: A reused value may be changed by another thread in the background

```
1 // myArray is an array of non-zero integers
2 // located in global or shared memory
3 __global__ void myKernel(int* result)
4 {
5     int tid = threadIdx.x;
6     int ref1 = myArray[tid] * 1;
7     myArray[tid + 1] = 2;
8     int ref2 = myArray[tid] * 1;
9     result[tid] = ref1 * ref2;
10 }
```

- the first reference to myArray[tid] compiles into a memory read instruction

Volatile variables and specific compiler optimizations

- One of the compiler's tricks: reuse references to memory location
- Result: A reused value may be changed by another thread in the background

```
1 // myArray is an array of non-zero integers
2 // located in global or shared memory
3 __global__ void myKernel(int* result)
4 {
5     int tid = threadIdx.x;
6     int ref1 = myArray[tid] * 1;
7     myArray[tid + 1] = 2;
8     int ref2 = myArray[tid] * 1;
9     result[tid] = ref1 * ref2;
10 }
```

- the first reference to myArray[tid] compiles into a memory read instruction
- the second reference does not as the compiler simply reuses the ~~result~~ of the first read

Outline

4 Optimizations

- Basics
- Measurements
- Instruction Optimizations
- Memory optimizations
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Outline

4 Optimizations

• Basics

- Measurements
- Instruction Optimizations
- Memory optimizations
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Optimization Basics

- Maximize parallel blocks (many threads and many blocks)

Optimization Basics

- Maximize parallel blocks (many threads and many blocks)
- Maximize computations / minimize memory transfer

Optimization Basics

- Maximize parallel blocks (many threads and many blocks)
- Maximize computations / minimize memory transfer
- Avoid costly operations

Optimization Basics

- Maximize parallel blocks (many threads and many blocks)
- Maximize computations / minimize memory transfer
- Avoid costly operations
- Take advantage of shared memory

Outline

4 Optimizations

- Basics
- Measurements
- Instruction Optimizations
- Memory optimizations
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Timers API

- `cudaEvent_t` – event type
- `cudaEventSynchronize()` – blocks CPU until given event records
- `cudaEventRecord()` – records given event in given stream
- `cudaEventElapsedTime()` – calculates time in milliseconds between events
- `cudaEventCreate()` – creates an event
- `cudaEventDestroy()` – destroys an event

Timers Example

```
1 cudaEvent_t start, stop; float time;
2 cudaEventCreate(&start);
3 cudaEventCreate(&stop);
4
5 cudaEventRecord( start, 0 );
6
7 kernel<<<grid,threads>>> ( d_odata, d_idata, size_x, size_y, NUM_REPS );
8
9 cudaEventRecord( stop, 0 );
10
11 cudaEventSynchronize( stop );
12
13 cudaEventElapsedTime( &time, start, stop );
14
15 cudaEventDestroy( start );
16 cudaEventDestroy( stop );
```

Timers with cutill

```
1 unsigned int timer;
2 cutillCheckError( cutCreateTimer(&timer) );
3
4 cutStartTimer(timer);
5
6 [...]
7
8 cudaThreadSynchronize();
9
10 cutStopTimer(timer);
11
12 timer_result = cutGetTimerValue(timer);
13
14 cutResetTimer(timer);
15
16 cutillCheckError(cutDeleteTimer(timer));
```

Theoretical Bandwidth Calculation

$$TB = (\text{Clock} \times 10^6 \times \text{MemInt} \times 2) / 10^9$$

- TB – theoretical bandwidth [GB/s]
- Clock – memory clock rate [MHz]
- MemInt – width of memory interface [B]
- 2 – DDR – Double Data Rate Memory

Theoretical Bandwidth Calculation

$$TB = (Clock \times 10^6 \times MemInt \times 2) / 10^9$$

- TB – theoretical bandwidth [GB/s]
- Clock – memory clock rate [MHz]
- MemInt – width of memory interface [B]
- 2 – DDR – Double Data Rate Memory

For NVIDIA GeForce GTX 280 we get:

$$(1107 \times 10^6 \times 512/8 \times 2) / 10^9 = 141.6 \text{ [GB/s]}$$

Effective Bandwidth Calculation

$$EB = \frac{(B_r + B_w) \times 10^{-9}}{t}$$

- EB – effective bandwidth [GB/s]
- B_r – bytes read
- B_w – bytes written
- t – time of the test [s]

Outline

4 Optimizations

- Basics
- Measurements
- Instruction Optimizations
- Memory optimizations
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Instruction Optimizations

- Use faster equivalents for slower ones
- Use single precision (functions, variables and constants)
- Use specialized math functions

Find balance between speed and precision

Functions working in host and device code (selection):

single prec.: $x+y$, $x*y$, x/y , $1/x$, $\text{sqrtf}(x)$, $\text{rsqrtf}(x)$, $\text{expf}(x)$, $\text{exp2f}(x)$,
 $\text{exp10f}(x)$, $\text{logf}(x)$, $\text{log2f}(x)$, $\text{log10f}(x)$, $\text{sinf}(x)$, $\text{cosf}(x)$,
 $\text{tanf}(x)$, $\text{asinf}(x)$, $\text{acosf}(x)$, $\text{atanf}(x)$, $\text{sinhf}(x)$, ...
 $\text{powf}(x,y)$
 $\text{rintf}(x)$, $\text{truncf}(x)$, $\text{ceilf}(x)$, $\text{floorf}(x)$,
 $\text{roundf}(x)$

double prec.: $x+y$, $x*y$, x/y , $1/x$, $\text{sqrt}(x)$, $\text{rsqrt}(x)$, $\text{exp}(x)$, $\text{exp2}(x)$,
 $\text{exp10}(x)$, $\text{log}(x)$, $\text{log2}(x)$, $\text{log10}(x)$, $\text{sin}(x)$, $\text{cos}(x)$,
 $\text{tan}(x)$, $\text{asin}(x)$, $\text{acos}(x)$, $\text{atan}(x)$, $\text{sinh}(x)$, ...
 $\text{pow}(x,y)$,
 $\text{rint}(x)$, $\text{trunc}(x)$, $\text{ceil}(x)$, $\text{floor}(x)$,
 $\text{round}(x)$

integer func.: $\text{min}(x,y)$, $\text{max}(x,y)$

When executed in host code, a given function uses the C runtime
implementation

Several code samples

If the following code is executed on a device with $cc \leq 1.2$ or on device with $cc > 1.2$ but with disabled double precision

```
1 float a;  
2 a = a*1.02;
```

then multiplication is done in single precision.

Several code samples

If the following code is executed on a device with $cc \leq 1.2$ or on device with $cc > 1.2$ but with disabled double precision

```
1 float a;  
2 a = a*1.02;
```

then multiplication is done in single precision. What happens if the same code is run on host?

Several code samples

If the following code is executed on a device with $cc \leq 1.2$ or on device with $cc > 1.2$ but with disabled double precision

```
1 float a;  
2 a = a*1.02;
```

then multiplication is done in single precision. What happens if the same code is run on host? Multiplication is done in double precision.

Several code samples

If the following code is executed on a device with $cc \leq 1.2$ or on device with $cc > 1.2$ but with disabled double precision

```
1 float a;  
2 a = a*1.02;
```

then multiplication is done in single precision. What happens if the same code is run on host? Multiplication is done in double precision.

```
1 float a;  
2 a = a*1.02f;
```

Several code samples

If the following code is executed on a device with $cc \leq 1.2$ or on device with $cc > 1.2$ but with disabled double precision

```
1 float a;  
2 a = a*1.02;
```

then multiplication is done in single precision. What happens if the same code is run on host? Multiplication is done in double precision.

```
1 float a;  
2 a = a*1.02f;
```

```
1 sizeof(char)  = 1  
2 sizeof(short) = 2  
3 sizeof(int)   = 4  
4 sizeof(float) = 4  
5 sizeof(double)= 8
```

Device intrinsics mathematical functions

- Less accurate but faster device-only versions (selection):

```
__fadd_[rn,rz,ru,rd](x,y), __fmul_[rn,rz,ru,rd](x,y) __pow(x,y) __log(x),  
__log2(x), __log10(x) __exp(x) __sin(x), __cos(x), __tan(x) __umul24,  
__float2int_[rn,rz,ru,rd](x), __float2uint_[rn,rz,ru,rd](x)
```

- Double precision (selection):

```
__dadd_[rn,rz,ru,rd](x,y), __dmul_[rn,rz,ru,rd](x,y), __double2float_[rn  
,rz](x), __double2int_[rn,rz,ru,rd](x), __double2uint_[rn,rz,ru,rd](x)
```

Outline

4 Optimizations

- Basics
- Measurements
- Instruction Optimizations
- **Memory optimizations**
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Using different memory types

- On-chip registers – very limited – local kernel variables
- On-chip constant cache – 8kB – read by kernel
- On-chip shared memory – allocated by device or host – accessed in kernel
- Texture memory – allocated by host – read by kernel
- Global device memory – allocated by host – accessed in kernel and host

Using host page-locked memory

- (“pinned”) host memory enables highest `cudaMemcpy` performance

Using host page-locked memory

- (“pinned”) host memory enables highest `cudaMemcpy` performance
 - 3.2 GB/s on PCI-e x16 Gen1

Using host page-locked memory

- (“pinned”) host memory enables highest `cudaMemcpy` performance
 - 3.2 GB/s on PCI-e x16 Gen1
 - 5.2 GB/s on PCI-e x16 Gen2

Using host page-locked memory

- (“pinned”) host memory enables highest `cudaMemcpy` performance
 - 3.2 GB/s on PCI-e x16 Gen1
 - 5.2 GB/s on PCI-e x16 Gen2
- allows for parallel execution of data transfer and kernel

Using host page-locked memory

- (“pinned”) host memory enables highest `cudaMemcpy` performance
 - 3.2 GB/s on PCI-e x16 Gen1
 - 5.2 GB/s on PCI-e x16 Gen2
- allows for parallel execution of data transfer and kernel
- on some devices page-locked memory may be mapped directly to device’s memory

Using host page-locked memory

- (“pinned”) host memory enables highest `cudaMemcpy` performance
 - 3.2 GB/s on PCI-e x16 Gen1
 - 5.2 GB/s on PCI-e x16 Gen2
- allows for parallel execution of data transfer and kernel
- on some devices page-locked memory may be mapped directly to device’s memory
- Allocating too much page-locked memory can reduce overall system performance

Page-locked memory allocation |

- `cudaError_t cudaMallocHost (void **ptr, size_t size)`
- `cudaError_t cudaHostAlloc (void **ptr, size_t size,
unsigned int flags)`

Page-locked memory allocation I

- `cudaError_t cudaMallocHost (void **ptr, size_t size)`
- `cudaError_t cudaHostAlloc (void **ptr, size_t size,
unsigned int flags)`
- possible flags (yes, they are orthogonal):

Page-locked memory allocation I

- `cudaError_t cudaMallocHost (void **ptr, size_t size)`
- `cudaError_t cudaHostAlloc (void **ptr, size_t size, unsigned int flags)`
- possible flags (yes, they are orthogonal):
 - `cudaHostAllocDefault` – This flag's value is defined to be 0 and causes `cudaHostAlloc()` to emulate `cudaMallocHost()`.

Page-locked memory allocation I

- `cudaError_t cudaMallocHost (void **ptr, size_t size)`
- `cudaError_t cudaHostAlloc (void **ptr, size_t size, unsigned int flags)`
- possible flags (yes, they are orthogonal):
 - `cudaHostAllocDefault` – This flag's value is defined to be 0 and causes `cudaHostAlloc()` to emulate `cudaMallocHost()`.
 - `cudaHostAllocPortable` – The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.

Page-locked memory allocation I

- `cudaError_t cudaMallocHost (void **ptr, size_t size)`
- `cudaError_t cudaHostAlloc (void **ptr, size_t size, unsigned int flags)`
- possible flags (yes, they are orthogonal):
 - `cudaHostAllocDefault` – This flag's value is defined to be 0 and causes `cudaHostAlloc()` to emulate `cudaMallocHost()`.
 - `cudaHostAllocPortable` – The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
 - `cudaHostAllocMapped` – Maps the allocation into the CUDA address space.

Page-locked memory allocation |

- `cudaError_t cudaMallocHost (void **ptr, size_t size)`
- `cudaError_t cudaHostAlloc (void **ptr, size_t size, unsigned int flags)`
- possible flags (yes, they are orthogonal):
 - `cudaHostAllocDefault` – This flag's value is defined to be 0 and causes `cudaHostAlloc()` to emulate `cudaMallocHost()`.
 - `cudaHostAllocPortable` – The memory returned by this call will be considered as pinned memory by all CUDA contexts, not just the one that performed the allocation.
 - `cudaHostAllocMapped` – Maps the allocation into the CUDA address space.
 - `cudaHostAllocWriteCombined` – Allocates the memory as write-combined (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs.

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.
 - `cudaSetDeviceFlags()` – flag: `cudaDeviceMapHost`

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.
 - `cudaSetDeviceFlags()` – flag: `cudaDeviceMapHost`
- Memory allocated by this function must be freed with `cudaFreeHost()`.

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.
 - `cudaSetDeviceFlags()` – flag: `cudaDeviceMapHost`
- Memory allocated by this function must be freed with `cudaFreeHost()`.
- Real life:

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.
 - `cudaSetDeviceFlags()` – flag: `cudaDeviceMapHost`
- Memory allocated by this function must be freed with `cudaFreeHost()`.
- Real life:
 - Pinned memory may double memory transfer if used properly

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.
 - `cudaSetDeviceFlags()` – flag: `cudaDeviceMapHost`
- Memory allocated by this function must be freed with `cudaFreeHost()`.
- Real life:
 - Pinned memory may double memory transfer if used properly
 - For some devices mapped host memory may save on memory transfer ('zero-copy' function is available since CUDA 2.2)

Page-locked memory allocation II

- The device pointer to the memory may be obtained by calling `cudaHostGetDevicePointer()`
- Before one may get device pointer to mapped host memory it must be enabled in driver.
 - `cudaSetDeviceFlags()` – flag: `cudaDeviceMapHost`
- Memory allocated by this function must be freed with `cudaFreeHost()`.
- Real life:
 - Pinned memory may double memory transfer if used properly
 - For some devices mapped host memory may save on memory transfer ('zero-copy' function is available since CUDA 2.2)
 - Write-combined memory reported to have no influence for current hardware

Global Memory alignment

- A device is capable of reading 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction.

```
1 __device__ type device[32];
2 type data = device[tid];
```

- `type` must be of size 4, 8, or 16
- its variables structures must be aligned:

```
1 struct __align__(16) {
2     float a;
3     float b;
4     float c;
5 };
```

One load instruction

Global Memory alignment

- A device is capable of reading 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction.

```
1 __device__ type device[32];
2 type data = device[tid];
```

- `type` must be of size 4, 8, or 16
- its variables structures must be aligned:

```
1 struct __align__(16) {
2     float a;
3     float b;
4     float c;
5 };
```

One load instruction

```
1 struct __align__(16) {
2     float a;
3     float b;
4     float c;
5     float d;
6     float e;
7 };
```

Two load instructions instead of five

Coalesced memory access

- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads) .

Coalesced memory access

- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads) .
- All read may result in one or two memory accesses.

Coalesced memory access

- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads) .
- All read may result in one or two memory accesses.
- If threads do not fulfil additional conditions reads are serialized resulting in 16 or more memory accesses.

Coalesced memory access

- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads) .
- All read may result in one or two memory accesses.
- If threads do not fulfil additional conditions reads are serialized resulting in 16 or more memory accesses.
- Coalescing is possible for segments of 32, 64 and 128 bytes

Coalesced memory access

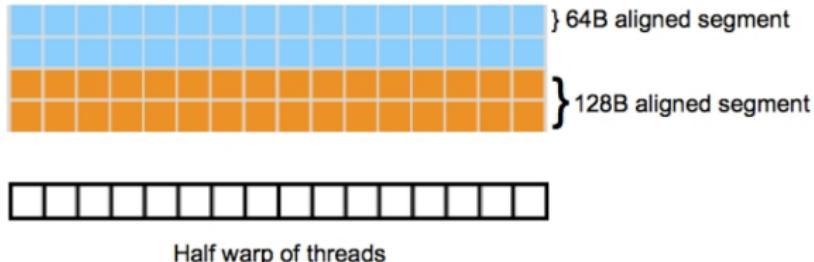
- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads) .
- All read may result in one or two memory accesses.
- If threads do not fulfil additional conditions reads are serialized resulting in 16 or more memory accesses.
- Coalescing is possible for segments of 32, 64 and 128 bytes
- Starting address for a region must be a multiple of region size

Coalesced memory access

- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads) .
- All read may result in one or two memory accesses.
- If threads do not fulfil additional conditions reads are serialized resulting in 16 or more memory accesses.
- Coalescing is possible for segments of 32, 64 and 128 bytes
- Starting address for a region must be a multiple of region size

Coalesced memory access

- **Coalesced global memory access** – A coordinated read by a half-warp (16 threads).
- All read may result in one or two memory accesses.
- If threads do not fulfil additional conditions reads are serialized resulting in 16 or more memory accesses.
- Coalescing is possible for segments of 32, 64 and 128 bytes
- Starting address for a region must be a multiple of region size



Coalesced memory access - devices with $cc \leq 1.1$

- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)

Coalesced memory access - devices with $cc \leq 1.1$

- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)
- A contiguous region of global memory:

Coalesced memory access - devices with $cc \leq 1.1$

- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)
- A contiguous region of global memory:
 - 64 bytes – each thread reads a word: `int`, `float`, ...

Coalesced memory access - devices with $cc \leq 1.1$

- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)
- A contiguous region of global memory:
 - 64 bytes – each thread reads a word: `int`, `float`, ...
 - 128 bytes – each thread reads a double-word: `int2`, `float2`, ...

Coalesced memory access - devices with $cc \leq 1.1$

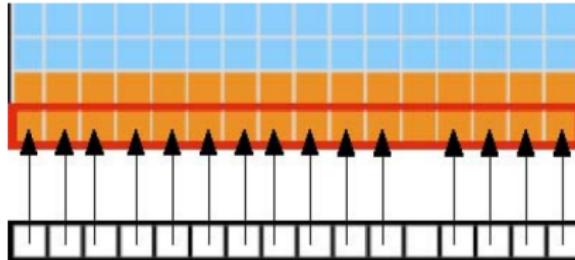
- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)
- A contiguous region of global memory:
 - 64 bytes – each thread reads a word: `int`, `float`, ...
 - 128 bytes – each thread reads a double-word: `int2`, `float2`, ...
 - 256 bytes – each thread reads a quad-word: `int4`, `float4`, ...
(two memory transactions)

Coalesced memory access - devices with $cc \leq 1.1$

- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)
- A contiguous region of global memory:
 - 64 bytes – each thread reads a word: `int`, `float`, ...
 - 128 bytes – each thread reads a double-word: `int2`, `float2`, ...
 - 256 bytes – each thread reads a quad-word: `int4`, `float4`, ...
(two memory transactions)

Coalesced memory access - devices with $cc \leq 1.1$

- The kth thread in a half-warp must access the kth element in a block being read
(not all threads must be participating)
- A contiguous region of global memory:
 - 64 bytes – each thread reads a word: `int`, `float`, ...
 - 128 bytes – each thread reads a double-word: `int2`, `float2`, ...
 - 256 bytes – each thread reads a quad-word: `int4`, `float4`, ...
(two memory transactions)



Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:

Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,

Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,
 - 64 bytes if all threads access 2-byte words,

Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,
 - 64 bytes if all threads access 2-byte words,
 - 128 bytes if all threads access 4-byte or 8-byte words.

Coalesced memory access - devices with $cc \geq 1.2$

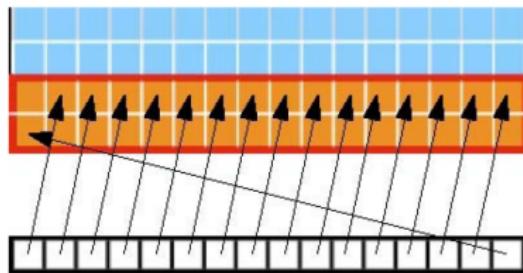
- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,
 - 64 bytes if all threads access 2-byte words,
 - 128 bytes if all threads access 4-byte or 8-byte words.
- Coalescing is achieved for any pattern of addresses requested by the half-warp, including patterns where multiple threads access the same address.

Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,
 - 64 bytes if all threads access 2-byte words,
 - 128 bytes if all threads access 4-byte or 8-byte words.
- Coalescing is achieved for any pattern of addresses requested by the half-warp, including patterns where multiple threads access the same address.

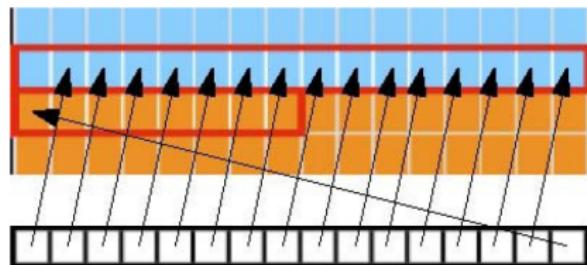
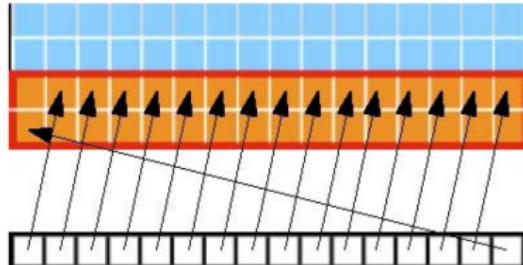
Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,
 - 64 bytes if all threads access 2-byte words,
 - 128 bytes if all threads access 4-byte or 8-byte words.
- Coalescing is achieved for any pattern of addresses requested by the half-warp, including patterns where multiple threads access the same address.



Coalesced memory access - devices with $cc \geq 1.2$

- Memory accesses are coalesced into a single memory transaction if words accessed by all threads lie in the same segment of size equal to:
 - 32 bytes if all threads access 1-byte words,
 - 64 bytes if all threads access 2-byte words,
 - 128 bytes if all threads access 4-byte or 8-byte words.
- Coalescing is achieved for any pattern of addresses requested by the half-warp, including patterns where multiple threads access the same address.



Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:
 - $356\mu s$ – coalesced

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:
 - $356\mu s$ – coalesced
 - $357\mu s$ – coalesced, some threads don't participate

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:
 - $356\mu s$ – coalesced
 - $357\mu s$ – coalesced, some threads don't participate
 - $3,494\mu s$ – permuted/misaligned thread access

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:
 - $356\mu s$ – coalesced
 - $357\mu s$ – coalesced, some threads don't participate
 - $3,494\mu s$ – permuted/misaligned thread access
- 4K blocks x 256 threads reading float3s:

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:
 - $356\mu s$ – coalesced
 - $357\mu s$ – coalesced, some threads don't participate
 - $3,494\mu s$ – permuted/misaligned thread access
- 4K blocks x 256 threads reading float3s:
 - $3,302\mu s$ – float3 uncoalesced

Coalescing: Timing Results

Kernel: read a float, increment, write back

3M floats (12MB)

Times averaged over 10K runs

- 12K blocks x 256 threads reading floats:
 - $356\mu s$ – coalesced
 - $357\mu s$ – coalesced, some threads don't participate
 - $3,494\mu s$ – permuted/misaligned thread access
- 4K blocks x 256 threads reading float3s:
 - $3,302\mu s$ – float3 uncoalesced
 - $359\mu s$ – float3 coalesced through shared memory

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`
 - `type*` must meet the size and alignment requirements

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`
 - `type*` must meet the size and alignment requirements
 - if size of `type*` is larger than 16 it must be treated with additional care

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`
 - `type*` must meet the size and alignment requirements
 - if size of `type*` is larger than 16 it must be treated with additional care
- For two-dimensional arrays

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`
 - `type*` must meet the size and alignment requirements
 - if size of `type*` is larger than 16 it must be treated with additional care
- For two-dimensional arrays
 - array of `type*` accessed by `BaseAddress + width*tiy + tix`

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`
 - `type*` must meet the size and alignment requirements
 - if size of `type*` is larger than 16 it must be treated with additional care
- For two-dimensional arrays
 - array of `type*` accessed by `BaseAddress + width*tiy + tix`
 - `width` is a multiply of 16

Coalescing: Guidelines I

- Align data to fit equal segments in memory
(arrays allocated with `cudaMalloc...` are positioned to appropriate addresses automatically)
- For single-dimensional arrays
 - array of `type*` accessed by `BaseAddress + tid`
 - `type*` must meet the size and alignment requirements
 - if size of `type*` is larger than 16 it must be treated with additional care
- For two-dimensional arrays
 - array of `type*` accessed by `BaseAddress + width*tiy + tix`
 - `width` is a multiply of 16
 - The width of the thread block is a multiple of half the warp size

Coalescing: Guidelines II

- If proper memory alignment is impossible:

Coalescing: Guidelines II

- If proper memory alignment is impossible:
 - Use structures of arrays instead of arrays of structures

Coalescing: Guidelines II

- If proper memory alignment is impossible:
 - Use structures of arrays instead of arrays of structures

Coalescing: Guidelines II

- If proper memory alignment is impossible:
 - Use structures of arrays instead of arrays of structures

AoS	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
SoA	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4

Coalescing: Guidelines II

- If proper memory alignment is impossible:
 - Use structures of arrays instead of arrays of structures

AoS	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
SoA	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4

- Use `__align(4)`, `__align(8)` or `__align(16)` in structure declarations

Coalescing example I

cudaWhitpapers

Misaligned memory access with `float3` data

```
1 __global__ void accessFloat3(float3 *d_in, float3 d_out)
2 {
3     int index = blockIdx.x * blockDim.x + threadIdx.x;
4     float3 a = d_in[index];
5     a.x += 2;
6     a.y += 2;
7     a.z += 2;
8     d_out[index] = a;
9 }
```

- Each thread reads 3 floats = 12 bytes
- Half warp reads $16 * 12 = 192$ bytes
(three 64B non-contiguous segments)

Coalescing example II

Coalesced memory access with `float3` data

cudaWhitpapers

```
1 __global__ void accessFloat3Shared(float *g_in, float *g_out)
2 {
3     int index = 3 * blockIdx.x * blockDim.x + threadIdx.x;
4     __shared__ float s_data[256*3];
5     s_data[threadIdx.x] = g_in[index];
6     s_data[threadIdx.x+256] = g_in[index+256];
7     s_data[threadIdx.x+512] = g_in[index+512];
8     __syncthreads();
9     float3 a = ((float3*)s_data)[threadIdx.x];
10    a.x += 2;
11    a.y += 2;
12    a.z += 2;
13    ((float3*)s_data)[threadIdx.x] = a;
14    __syncthreads();
15    g_out[index] = s_data[threadIdx.x];
16    g_out[index+256] = s_data[threadIdx.x+256];
17    g_out[index+512] = s_data[threadIdx.x+512];
18 }
```

Computation part remains the same

Organization of shared memory

- Shared memory is divided into equally sized memory modules, called **banks**.

Organization of shared memory

- Shared memory is divided into equally sized memory modules, called **banks**.
- Different banks can be accessed simultaneously.

Organization of shared memory

- Shared memory is divided into equally sized memory modules, called **banks**.
- Different banks can be accessed simultaneously.
- Read or write to n addresses in n banks multiplies bandwidth of a single bank by n .

Organization of shared memory

- Shared memory is divided into equally sized memory modules, called **banks**.
- Different banks can be accessed simultaneously.
- Read or write to n addresses in n banks multiplies bandwidth of a single bank by n .
- If many threads refers the same bank the access is serialized – hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary.

Organization of shared memory

- Shared memory is divided into equally sized memory modules, called **banks**.
- Different banks can be accessed simultaneously.
- Read or write to n addresses in n banks multiplies bandwidth of a single bank by n .
- If many threads refers the same bank the access is serialized – hardware splits a memory request that has bank conflicts into as many separate conflict-free requests as necessary.
- There is one exception if all threads within a half-warp accesses the same address.

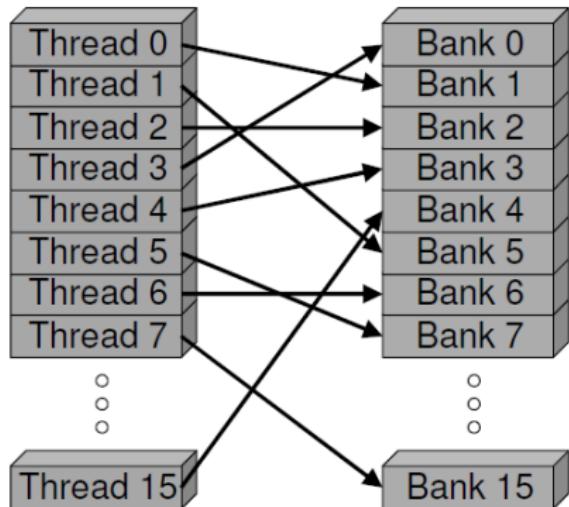
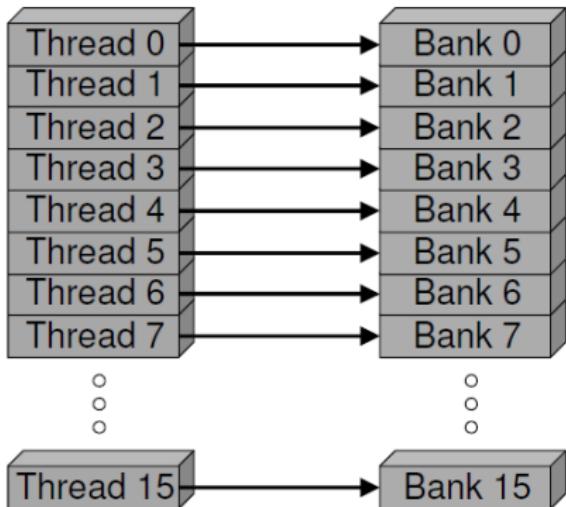
Bank conflicts

Shared memory banks are organized in such a way that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per clock cycle. The bandwidth of shared memory is 32 bits per bank per clock cycle.

Important

For devices of compute capability 1.x, the warp size is 32 threads and the number of banks is 16.

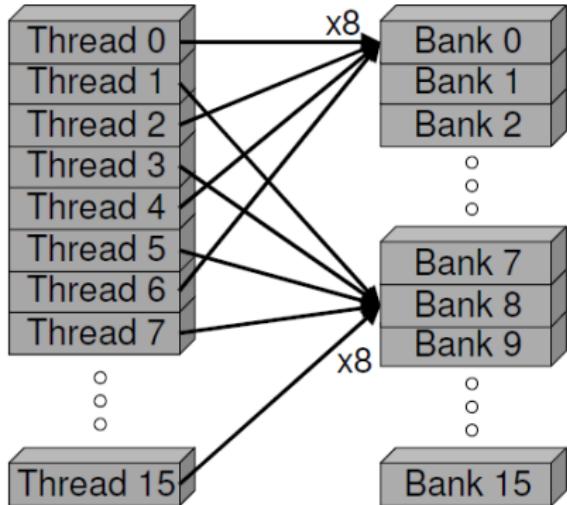
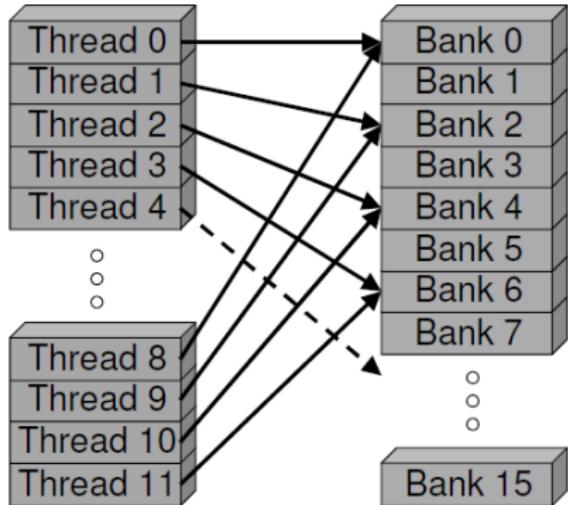
Access with no bank conflicts



left: stride = 1

right: stride random

Access with bank conflicts



left: stride = 2 (2 way bank conflict)

right: stride = 8 (8 way bank conflict)

Avoiding bank conflicts

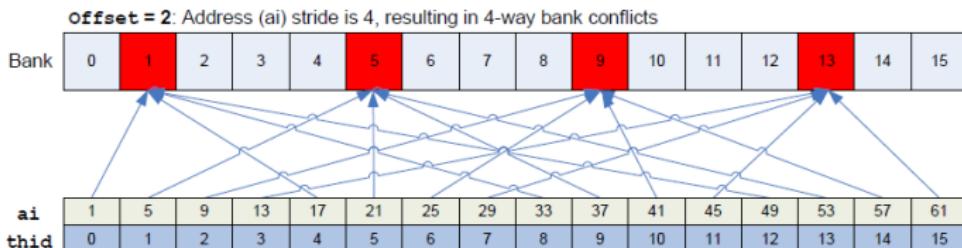
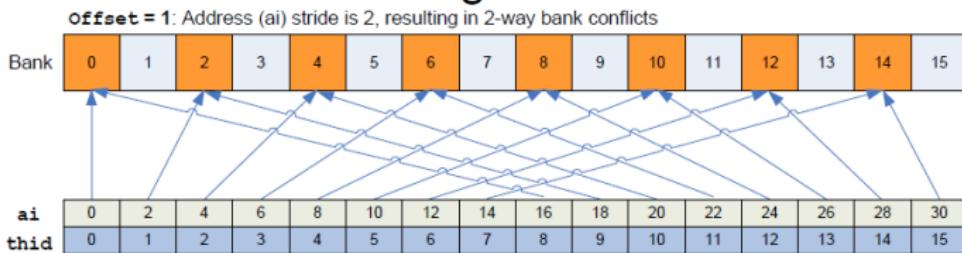
- Bank checker macro – included in SDK (common) – not very reliable
- Padding – adding extra space between array elements in order to break cyclic access to same bank.

Example of bank conflicts removal in Scan I

Harris_scanCuda

Addressing Without Padding

```
int ai = offset*(2*thid+1)-1;  
int bi = offset*(2*thid+2)-1;  
temp[bi] += temp[ai];
```



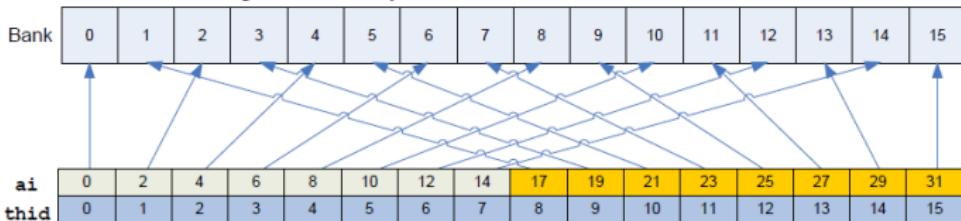
Example of bank conflicts removal in Scan II

Addressing With Padding

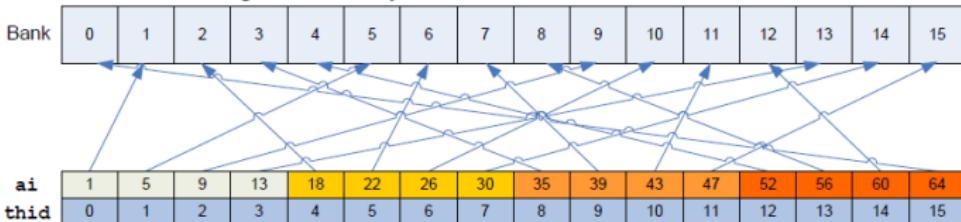
Harris_scanCuda

```
int ai = offset*(2*thid+1)-1;  
int bi = offset*(2*thid+2)-1;  
ai += ai / NUM_BANKS;  
bi += bi / NUM_BANKS;  
temp[bi] += temp[ai];
```

Offset = 1: Padding addresses every 16 elements removes bank conflicts



Offset = 2: Padding addresses every 16 elements removes bank conflicts



Padding implementation I

Harris_scanCuda

Padding implementation I

Harris_scanCuda

We need more space in shared memory:

```
1 unsigned int extra_space = num_elements / NUM_BANKS;
```

Padding implementation I

Harris_scanCuda

We need more space in shared memory:

```
1 unsigned int extra_space = num_elements / NUM_BANKS;
```

Padding macro:

```
1 #define NUM_BANKS 16
2 #define LOG_NUM_BANKS 4
3
4 #ifdef ZERO_BANK_CONFLICTS
5 #define CONFLICT_FREE_OFFSET(index) ((index) >> LOG_NUM_BANKS \
6                                     + (index) >> (2 * LOG_NUM_BANKS))
7 #else
8 #define CONFLICT_FREE_OFFSET(index) ((index) >> LOG_NUM_BANKS)
9 #endif
```

Padding implementation I

Harris_scanCuda

We need more space in shared memory:

```
1 unsigned int extra_space = num_elements / NUM_BANKS;
```

Padding macro:

```
1 #define NUM_BANKS 16
2 #define LOG_NUM_BANKS 4
3
4 #ifdef ZERO_BANK_CONFLICTS
5 #define CONFLICT_FREE_OFFSET(index) ((index) >> LOG_NUM_BANKS \
6                                     + (index) >> (2 * LOG_NUM_BANKS))
7 #else
8 #define CONFLICT_FREE_OFFSET(index) ((index) >> LOG_NUM_BANKS)
9 #endif
```

Zero bank conflicts requires even more additional space:

```
1 #ifdef ZERO_BANK_CONFLICTS
2     extra_space += extra_space / NUM_BANKS;
3 #endif
```

Padding implementation II

Harris_scanCuda

Padding implementation II

Harris_scanCuda

Loading data into shared memory:

```
1 int ai = thid, bi = thid + (n/2);
2
3 // compute spacing to avoid bank conflicts
4 int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
5 int bankOffsetB = CONFLICT_FREE_OFFSET(bi);
6
7 TEMP(ai + bankOffsetA) = g_idata[ai];
8 TEMP(bi + bankOffsetB) = g_idata[bi];
```

Padding implementation II

Harris_scanCuda

Loading data into shared memory:

```
1 int ai = thid, bi = thid + (n/2);
2
3 // compute spacing to avoid bank conflicts
4 int bankOffsetA = CONFLICT_FREE_OFFSET(ai);
5 int bankOffsetB = CONFLICT_FREE_OFFSET(bi);
6
7 TEMP(ai + bankOffsetA) = g_idata[ai];
8 TEMP(bi + bankOffsetB) = g_idata[bi];
```

Algorithm:

```
1 int ai = offset*(2*thid+1)-1;
2 int bi = offset*(2*thid+2)-1;
3
4 ai += CONFLICT_FREE_OFFSET(ai);
5 bi += CONFLICT_FREE_OFFSET(bi);
6
7 TEMP(bi) += TEMP(ai);
```

Outline

4 Optimizations

- Basics
- Measurements
- Instruction Optimizations
- Memory optimizations
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Streams API

- Applications manage concurrency through streams.

Streams API

- Applications manage concurrency through streams.
- A stream is a sequence of commands that execute in order.

Streams API

- Applications manage concurrency through streams.
- A stream is a sequence of commands that execute in order.
- Different streams may execute their commands out of order with respect to one another or concurrently.

Streams API

- Applications manage concurrency through streams.
- A stream is a sequence of commands that execute in order.
- Different streams may execute their commands out of order with respect to one another or concurrently.
- `cudaStream_t` – stream type

Streams API

- Applications manage concurrency through streams.
 - A stream is a sequence of commands that execute in order.
 - Different streams may execute their commands out of order with respect to one another or concurrently.
-
- `cudaStream_t` – stream type
 - `cudaStreamCreate(&stream)`

Streams API

- Applications manage concurrency through streams.
 - A stream is a sequence of commands that execute in order.
 - Different streams may execute their commands out of order with respect to one another or concurrently.
-
- `cudaStream_t` – stream type
 - `cudaStreamCreate(&stream)`
 - `cudaStreamDestroy(&stream)` – waits for all tasks to complete before destroying a stream;

Streams API

- Applications manage concurrency through streams.
 - A stream is a sequence of commands that execute in order.
 - Different streams may execute their commands out of order with respect to one another or concurrently.
-
- `cudaStream_t` – stream type
 - `cudaStreamCreate(&stream)`
 - `cudaStreamDestroy(&stream)` – waits for all tasks to complete before destroying a stream;
 - `cudaStreamQuery()` – checks if all preceding commands in a stream have completed

Streams API

- Applications manage concurrency through streams.
 - A stream is a sequence of commands that execute in order.
 - Different streams may execute their commands out of order with respect to one another or concurrently.
-
- `cudaStream_t` – stream type
 - `cudaStreamCreate(&stream)`
 - `cudaStreamDestroy(&stream)` – waits for all tasks to complete before destroying a stream;
 - `cudaStreamQuery()` – checks if all preceding commands in a stream have completed
 - `cudaStreamSynchronize()` – forces the run-time to wait until all preceding commands in a stream have completed.

Streams API

- Applications manage concurrency through streams.
 - A stream is a sequence of commands that execute in order.
 - Different streams may execute their commands out of order with respect to one another or concurrently.
-
- `cudaStream_t` – stream type
 - `cudaStreamCreate(&stream)`
 - `cudaStreamDestroy(&stream)` – waits for all tasks to complete before destroying a stream;
 - `cudaStreamQuery()` – checks if all preceding commands in a stream have completed
 - `cudaStreamSynchronize()` – forces the run-time to wait until all preceding commands in a stream have completed.
 - `cudaThreadSynchronize()` – forces the run-time to wait until all preceding device tasks in all streams have completed

Streams API – example

cudaProgrammingGuide

```
1 cudaStream_t stream[2];
2 for (int i = 0; i < 2; ++i)
3     cudaStreamCreate(&stream[i]);
4 float* hostPtr;
5 cudaMallocHost((void**)&hostPtr, 2 * size);
6 for (int i = 0; i < 2; ++i)
7     cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
8                     size, cudaMemcpyHostToDevice, stream[i]);
9 for (int i = 0; i < 2; ++i)
10    myKernel<<<100, 512, 0, stream[i]>>>
11        (outputDevPtr + i * size, inputDevPtr + i * size, size);
12 for (int i = 0; i < 2; ++i)
13     cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
14                     size, cudaMemcpyDeviceToHost, stream[i]);
15 cudaThreadSynchronize();
16 for (int i = 0; i < 2; ++i)
17     cudaStreamDestroy(&stream[i]);
```

Outline

4 Optimizations

- Basics
- Measurements
- Instruction Optimizations
- Memory optimizations
 - Playing with memory types
 - Coalesced memory access
 - Avoiding memory bank conflicts
- Asynchronous operations
 - Stream API
- Assuring optimum load for a device

Occupancy calculator I

- A tool for processors occupancy analysis

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:

Occupancy calculator |

- A tool for processors occupancy analysis
- Input:
 - Computational Capability

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:
 - Computational Capability
 - Threads per block

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:
 - Computational Capability
 - Threads per block
 - Registers per thread

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:
 - Computational Capability
 - Threads per block
 - Registers per thread
 - Shared memory per block (bytes)

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:
 - Computational Capability
 - Threads per block
 - Registers per thread
 - Shared memory per block (bytes)
- Output:

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:
 - Computational Capability
 - Threads per block
 - Registers per thread
 - Shared memory per block (bytes)
- Output:
 - GPU Occupancy

Occupancy calculator I

- A tool for processors occupancy analysis
- Input:
 - Computational Capability
 - Threads per block
 - Registers per thread
 - Shared memory per block (bytes)
- Output:
 - GPU Occupancy
 - Active blocks per multiprocessor

Occupancy calculator II

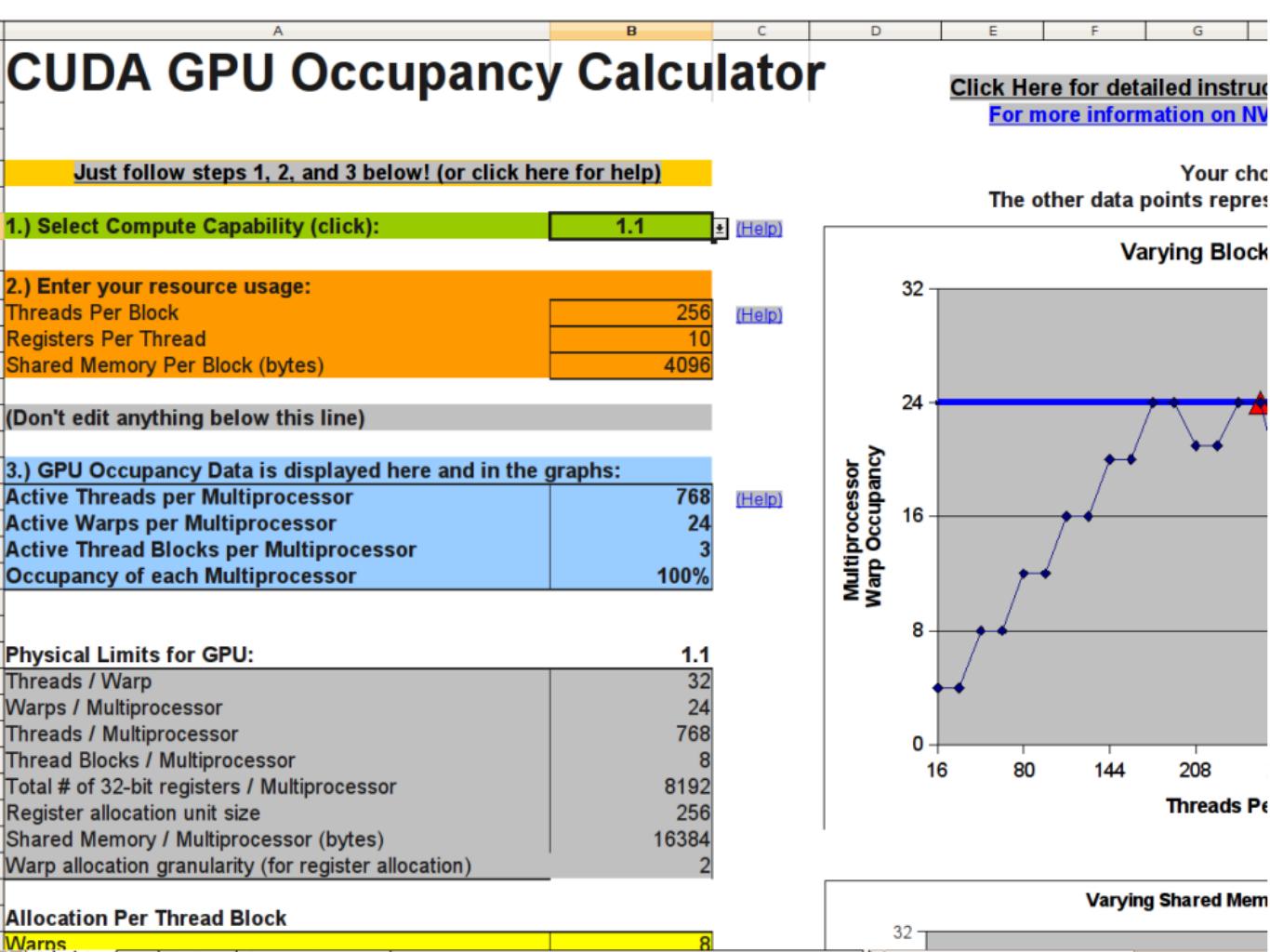
- Assuming a device with computational capability ≤ 1.1

Occupancy calculator II

- Assuming a device with computational capability ≤ 1.1
- Consider an application with 256 threads per block, 10 registers per thread, and 4KB of shared memory per thread block.

Occupancy calculator II

- Assuming a device with computational capability ≤ 1.1
- Consider an application with 256 threads per block, 10 registers per thread, and 4KB of shared memory per thread block.
- Then it can schedule 3 thread blocks and 768 threads on each SM.



Occupancy calculator III

Applying an optimization:

Increases each thread's register usage from 10 to 11
(an increase of only 10%)

Occupancy calculator III

Applying an optimization:

Increases each thread's register usage from 10 to 11
(an increase of only 10%)

- Decreases the number of blocks per SM from 3 to 2
- decreases the number of threads on an SM by 33% (Because of 8,192 registers limit per SM)

Ryo007programoptimization

CUDA GPU Occupancy Calculator

[Click Here for detailed instructions](#)

[For more information on NVIDIA](#)

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):

1.1

(Help)

2.) Enter your resource usage:

Threads Per Block

256

(Help)

Registers Per Thread

11

Shared Memory Per Block (bytes)

4096

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:

Active Threads per Multiprocessor

512

(Help)

Active Warps per Multiprocessor

16

Active Thread Blocks per Multiprocessor

2

Occupancy of each Multiprocessor

67%

Physical Limits for GPU:

1.1

Threads / Warp

32

Warps / Multiprocessor

24

Threads / Multiprocessor

768

Thread Blocks / Multiprocessor

8

Total # of 32-bit registers / Multiprocessor

8192

Register allocation unit size

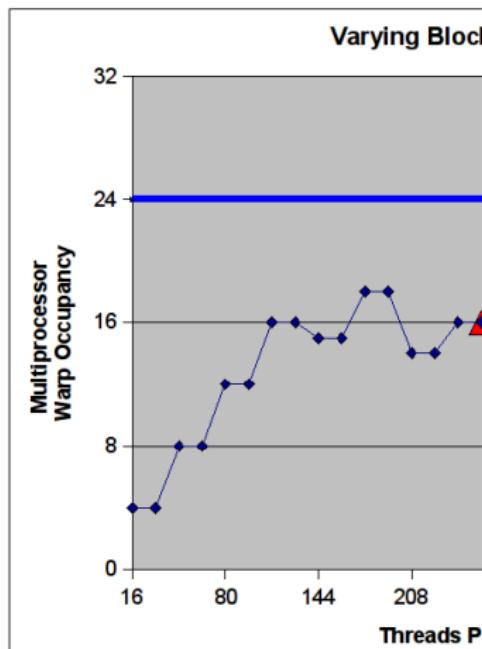
256

Shared Memory / Multiprocessor (bytes)

16384

Warp allocation granularity (for register allocation)

2



Device Computation Capabilities

Compute Capability	1.00	1.10	1.20	1.30	2.00	2.10	3.00	3.50
SM Version	sm_10	sm_11	sm_12	sm_13	sm_20	sm_21	sm_30	sm_35
Threads / Warp	32	32	32	32	32	32	32	32
Warps / Multiprocessor	24	24	32	32	48	48	64	64
Threads / Multiprocessor	768	768	1024	1024	1536	1536	2048	2048
Thread Blocks / Multiprocessor	8	8	8	8	8	8	16	16
Max Shared Memory / Multiprocessor [B]	16 384	16 384	16 384	16 384	49 152	49 152	49 152	49 152
Register File Size	8 192	8 192	16 384	16 384	32 768	32 768	65 536	65 536
Register Allocation Unit Size	256	256	512	512	64	64	256	256
Allocation Granularity	block	block	block	block	warp	warp	warp	warp
Max Registers / Thread	124	124	124	124	63	63	63	255
Shared Memory Allocation Unit Size	512	512	512	512	128	128	256	256
Warp allocation granularity	2	2	2	2	2	2	4	4
Max Thread Block Size	512	512	512	512	1 024	1 024	1 024	1 024
Shared Memory Size Configurations [B]	16 384	16 384	16 384	16 384	49 152	49 152	49 152	49 152

Bibliography