Setup

ssh to DAVINCI and grab a GPU node (if you don't already have one):

```
$ source /projects/k2i/hpc/scripts/login-gpu.sh
```

Once you have a GPU node, load the necessary packages and clone today's Github repo:

```
$ module load GCCcore/6.4.0 Git/2.17.1
$ source /projects/k2i/hpc/scripts/gpu-day-one-setup.sh
$ git clone https://github.com/agrippa/hpc-bootcamp.git $HOME/bootcamp-gpu
$ module unload Git/2.17.1 GCCcore/6.4.0
```

It also may be useful to have the CUDA API references open:

http://docs.nvidia.com/cuda/cuda-runtime-api/index.html

Slides are accessible in your browser in Github and Box:

Github: https://github.com/agrippa/hpc-bootcamp Box: http://bit.ly/hpc-bc19

GPU Accelerated Computing (Day 1)

Introduction to Programming GPUs

Max Grossman

Habanero Extreme Scale Software Research Group, Rice University

Principal & Co-Founder, 7pod Technologies

Author, Professional CUDA C Programming

Outline

1. A Brief History of Computing

2. GPU Architecture Overview

3. An Overview of CUDA

4. Hands-On with a Scientific App

5. Related Topics:
    1. Alternative GPU Programming Models
    2. Alternatives to GPUs

About Me

Habanero Research Group

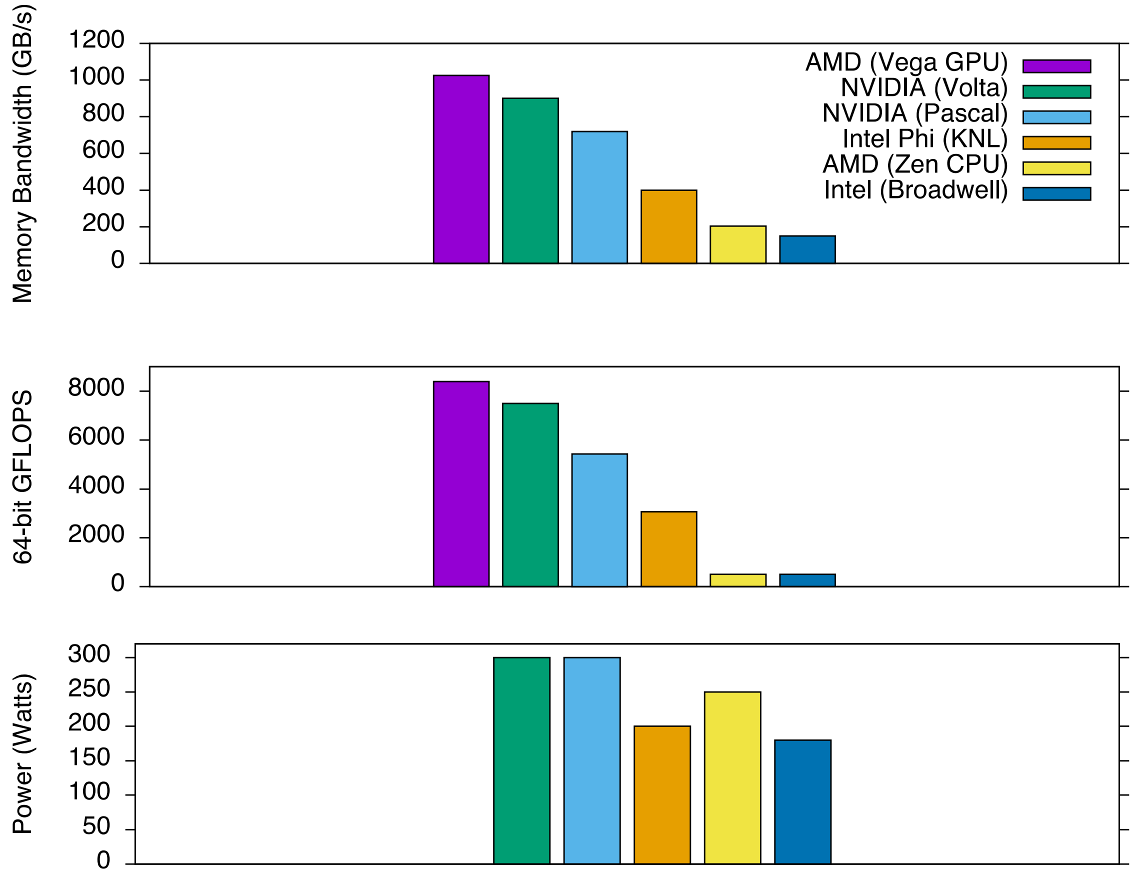
- Runtime Systems for HPC (“Big Compute”)
- Data Analytics (“Big Data”)
- Compilers and Programmer Tools

Co-Founder, 7pod Technologies (7pod.tech)

- Consulting on new applications of data analytics and high performance computing
- HPC and data analytics training

Why GPUs?

Upcoming Hardware Generations

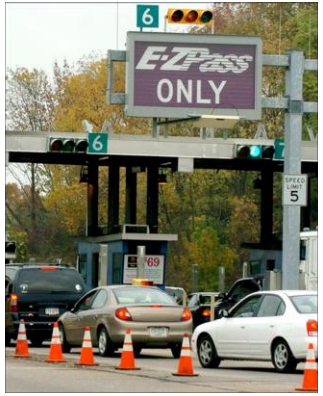


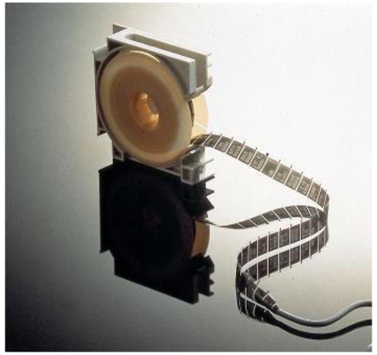
Is your hardware holding you back, or are you holding back your hardware?

A (Brief) History of Early Personal Computing

1971: Intel 4004

1974: Intel 8080

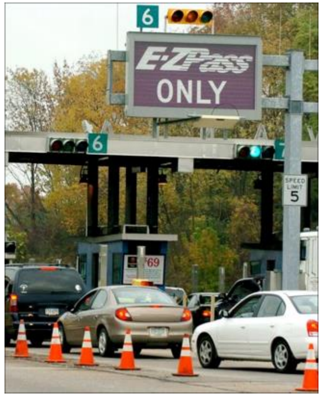


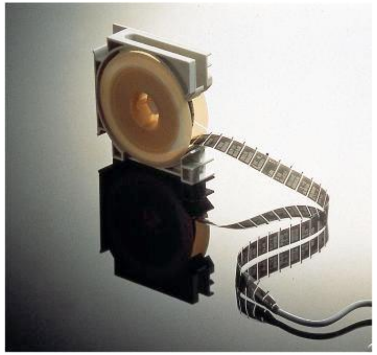


http://www.computerhope.com/history/processor.htm

A (Brief) History of Early Personal Computing

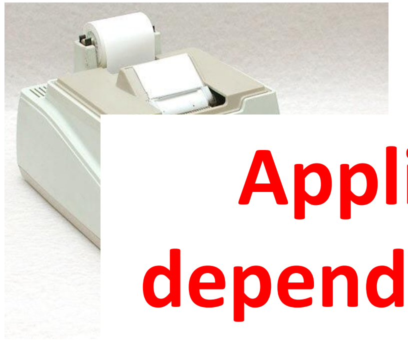
What do all of these applications have in common?

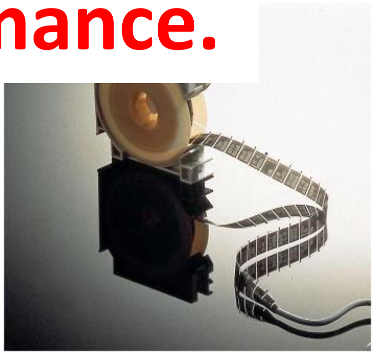




A (Brief) History of Early Personal Computing

What do all of these applications have in common?

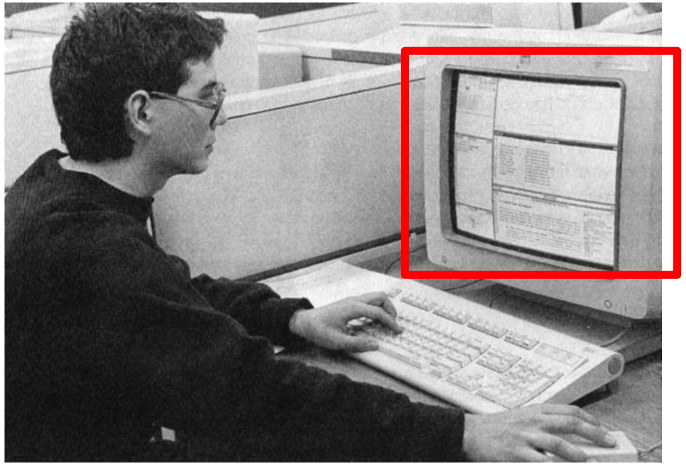




Application performance is entirely dependent on straight-line performance.

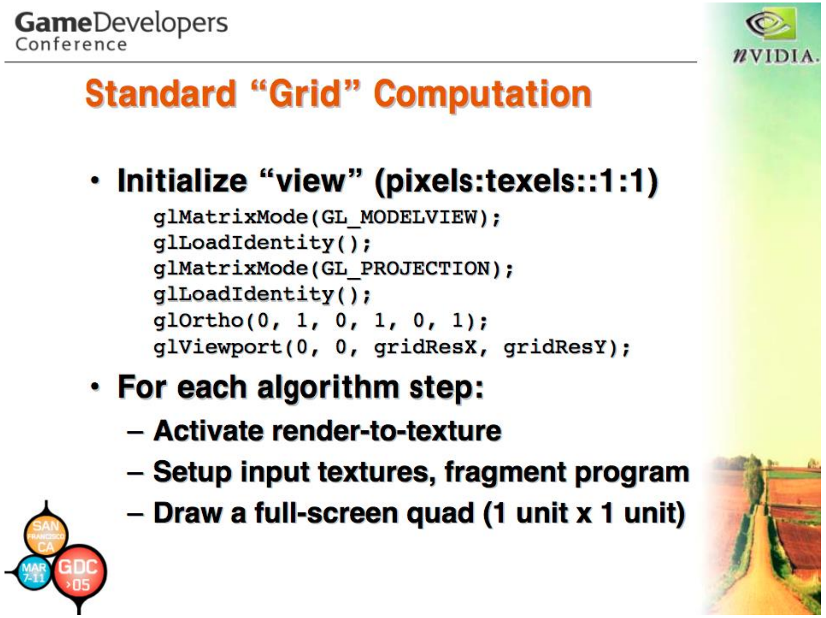
A (Brief) History of Early Personal Computing

First dedicated Graphics Processing Units (GPUs) introduced in the 1990s.



A (Brief) History of Early Personal Computing

Only became programmable in the early 2000s.



GameDevelopers Conference

NVIDIA.

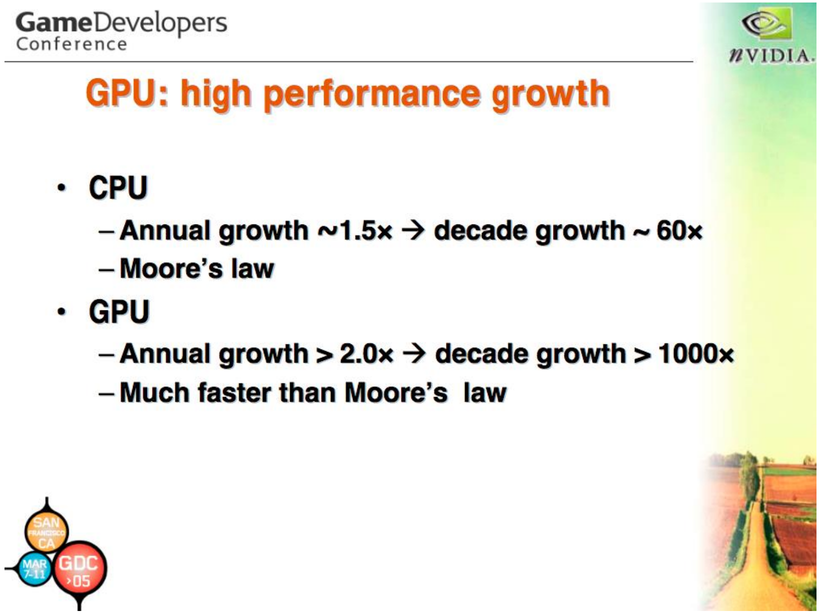
Standard "Grid" Computation

- Initialize "view" (pixels:texels::1:1)

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 1, 0, 1, 0, 1);
glViewport(0, 0, gridResX, gridResY);
```

- For each algorithm step:
  - Activate render-to-texture
  - Setup input textures, fragment program
  - Draw a full-screen quad (1 unit x 1 unit)

SAN FRANCISCO CA MAR 7-11 GDC 05



GameDevelopers Conference

NVIDIA.

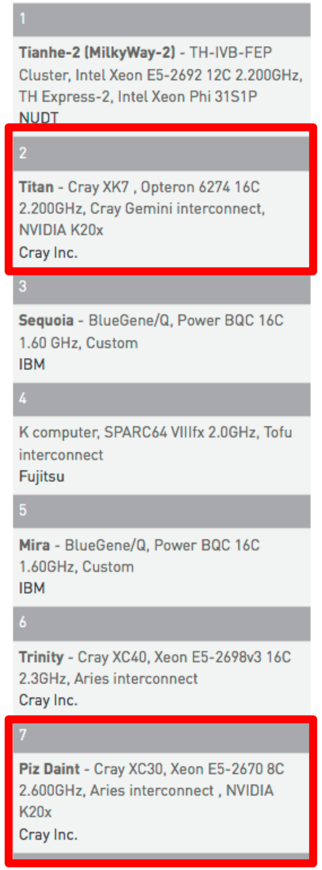
GPU: high performance growth

- CPU
  - Annual growth ~1.5× → decade growth ~ 60×
  - Moore's law
- GPU
  - Annual growth > 2.0× → decade growth > 1000×
  - Much faster than Moore's law

SAN FRANCISCO CA MAR 7-11 GDC 05

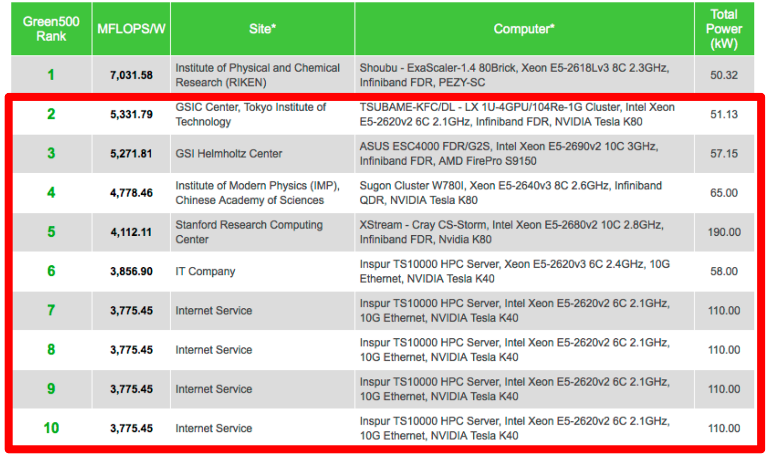
ftp://77.67.22.188/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_GPGPU.pdf

A (Brief) History of Early Personal Computing

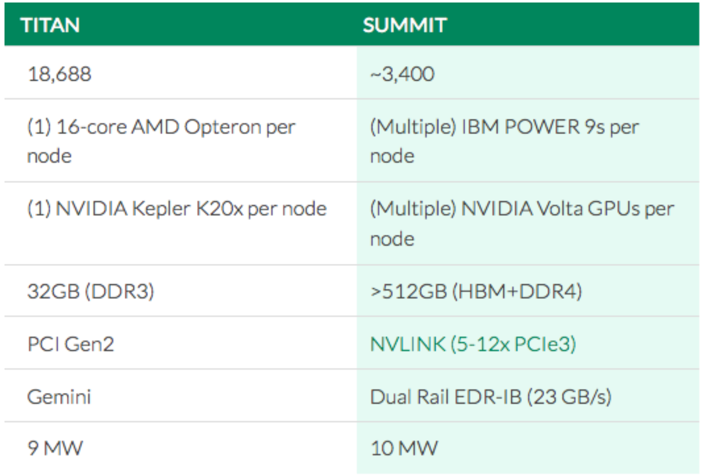


1
Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT
2
Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.
3
Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM
4
K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu
5
Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM
6
Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.
7
Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.

And a decade later…



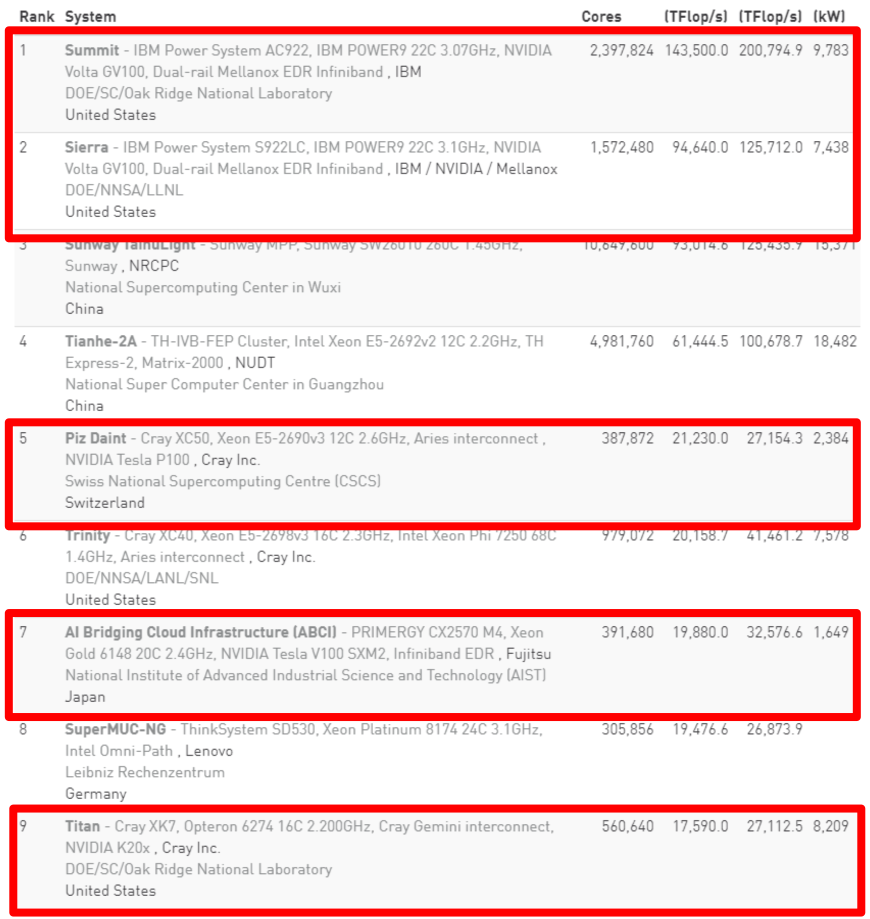
Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	7,031.58	Institute of Physical and Chemical Research (RIKEN)	Shoubu - ExaScaler-1.4 80Brick, Xeon E5-2618Lv3 8C 2.3GHz, Infiniband FDR, PEZY-SC	50.32
2	5,331.79	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC/DL - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.1GHz, Infiniband FDR, NVIDIA Tesla K80	51.13
3	5,271.81	GSI Helmholtz Center	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, AMD FirePro S9150	57.15
4	4,778.46	Institute of Modern Physics (IMP), Chinese Academy of Sciences	Sugon Cluster W780I, Xeon E5-2640v3 8C 2.6GHz, Infiniband QDR, NVIDIA Tesla K80	65.00
5	4,112.11	Stanford Research Computing Center	XStream - Cray CS-Storm, Intel Xeon E5-2680v2 10C 2.8GHz, Infiniband FDR, Nvidia K80	190.00
6	3,856.90	IT Company	Inspur TS10000 HPC Server, Xeon E5-2620v3 6C 2.4GHz, 10G Ethernet, NVIDIA Tesla K40	58.00
7	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00
8	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00
9	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00
10	3,775.45	Internet Service	Inspur TS10000 HPC Server, Intel Xeon E5-2620v2 6C 2.1GHz, 10G Ethernet, NVIDIA Tesla K40	110.00



TITAN	SUMMIT
18,688	~3,400
(1) 16-core AMD Opteron per node	(Multiple) IBM POWER 9s per node
(1) NVIDIA Kepler K20x per node	(Multiple) NVIDIA Volta GPUs per node
32GB (DDR3)	>512GB (HBM+DDR4)
PCI Gen2	NVLINK (5-12x PCIe3)
Gemini	Dual Rail EDR-IB (23 GB/s)
9 MW	10 MW

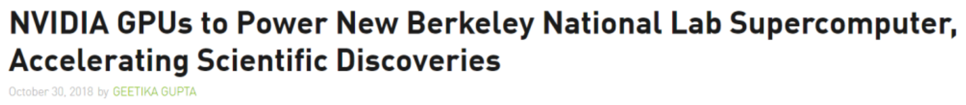
https://www.olcf.ornl.gov/summit/, http://top500.org/, http://www.green500.org/lists/green201511

A (Brief) History of Early Personal Computing



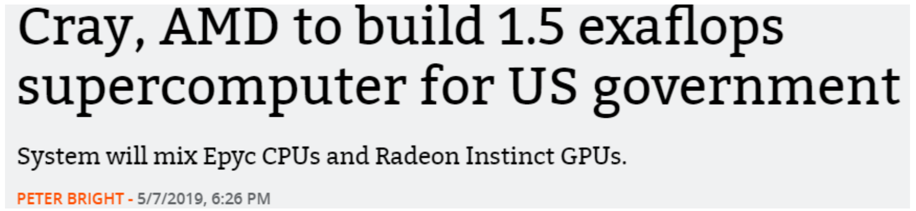
Rank	System	Cores	(TFlop/s)	(TFlop/s)	(kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	200,794.9	9,783
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	387,872	21,230.0	27,154.3	2,384
6	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,072	20,158.7	41,461.2	7,578
7	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
8	**SuperMUC-NG** - ThinkSystem SD530, Xeon Platinum 8174 24C 3.1GHz, Intel Omni-Path , Lenovo Leibniz Rechenzentrum Germany	305,856	19,476.6	26,873.9	
9	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209

Today's and Tomorrow's Top 500 list



NVIDIA GPUs to Power New Berkeley National Lab Supercomputer, Accelerating Scientific Discoveries

October 30, 2018 by GEETIKA GUPTA



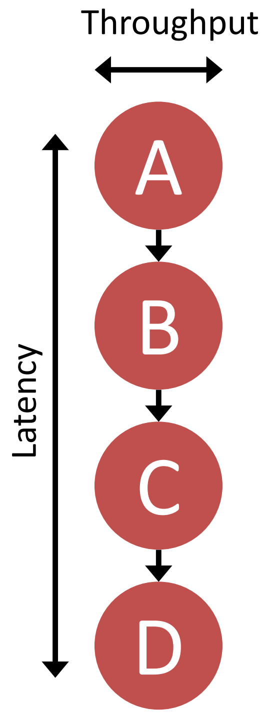
Cray, AMD to build 1.5 exaflops supercomputer for US government

System will mix Epyc CPUs and Radeon Instinct GPUs.

PETER BRIGHT - 5/7/2019, 6:26 PM

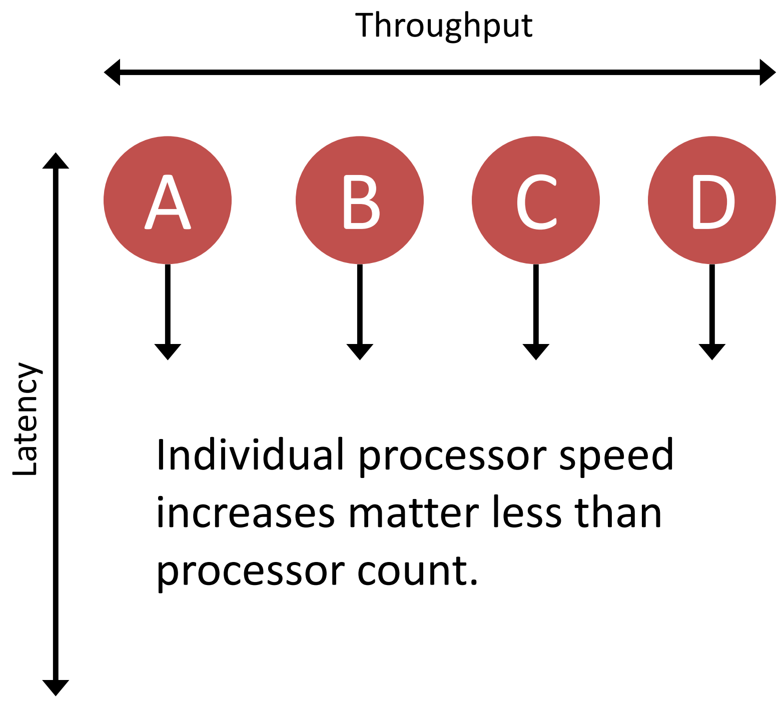
Latency vs. Throughput in Apps & Datasets

Latency-bound



Increasing the # of processors doesn't matter, only straight-line performance.

Throughput-bound



Individual processor speed increases matter less than processor count.

A spectrum, not a discrete distribution

Latency-Optimized Processors

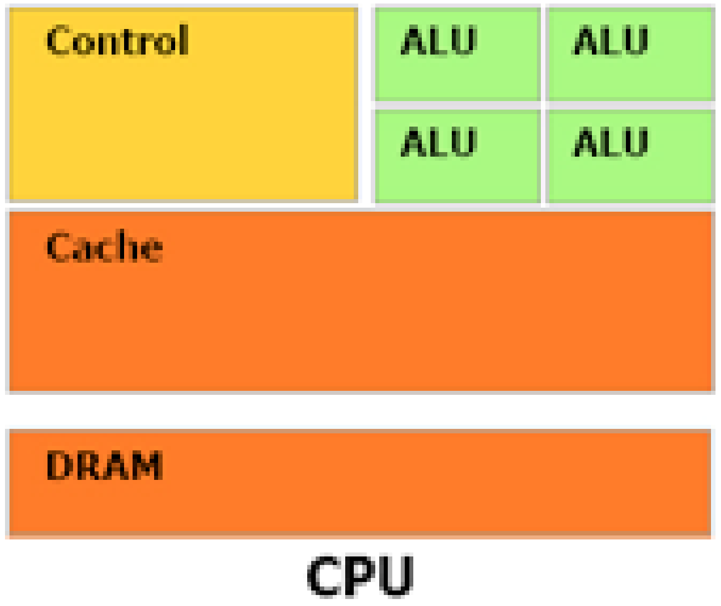
For most consumer (i.e. interactive) applications, latency is the #1 performance metric.

↑ clock frequencies

↑ control logic (e.g. branch prediction)

↑ cache per core

↓ parallelism/throughput



Throughput-Optimized Processors

For many scientific applications, throughput is the #1 performance metric.

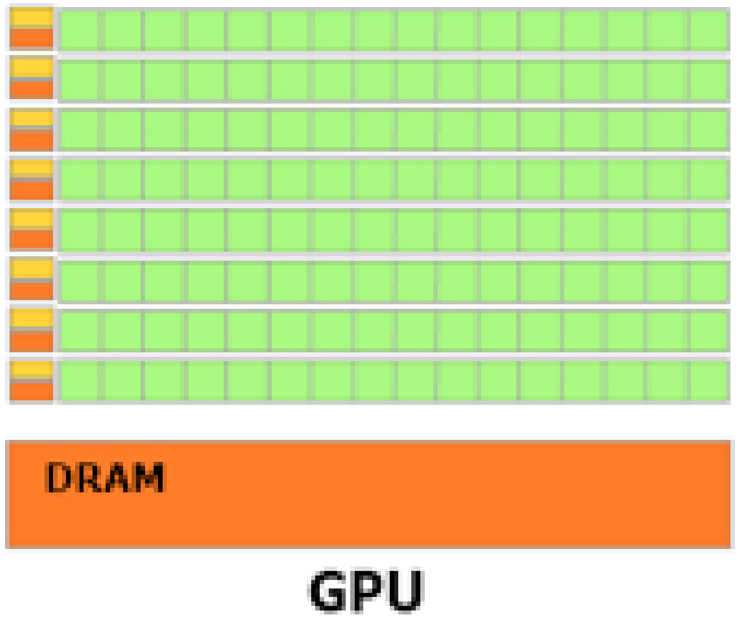
↑ cores/concurrency

↑ memory bandwidth

↓ clock frequency

↓ cache

↓ control logic



Throughput-Bound Applications

While many applications are latency-bound, the important ones (scientific, finance, medical, etc.) tend to be throughput bound.

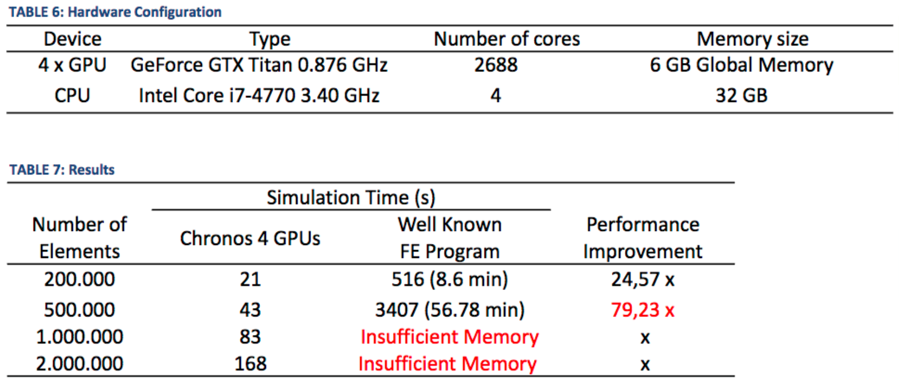


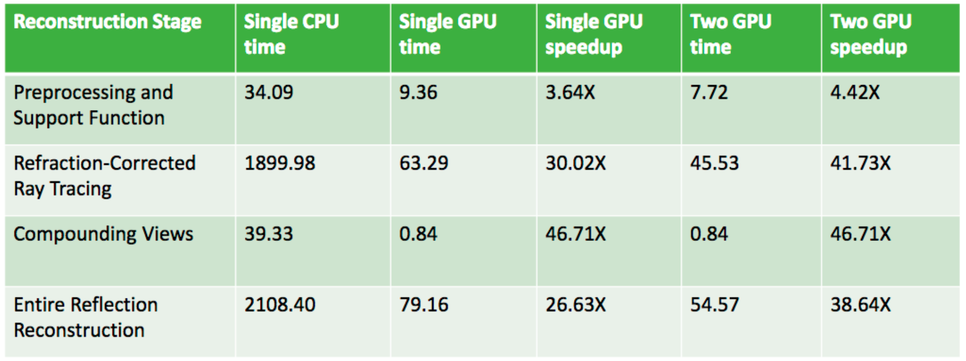
TABLE 6: Hardware Configuration

Device	Type	Number of cores	Memory size
4 x GPU	GeForce GTX Titan 0.876 GHz	2688	6 GB Global Memory
CPU	Intel Core i7-4770 3.40 GHz	4	32 GB

TABLE 7: Results

Number of Elements	Simulation Time (s) Chronos 4 GPUs	Simulation Time (s) Well Known FE Program	Performance Improvement
200.000	21	516 (8.6 min)	24,57 x
500.000	43	3407 (56.78 min)	79,23 x
1.000.000	83	Insufficient Memory	x
2.000.000	168	Insufficient Memory	x

“Speeding up a Finite Element Computation on GPU”, GTC 15



Reconstruction Stage	Single CPU time	Single GPU time	Single GPU speedup	Two GPU time	Two GPU speedup
Preprocessing and Support Function	34.09	9.36	3.64X	7.72	4.42X
Refraction-Corrected Ray Tracing	1899.98	63.29	30.02X	45.53	41.73X
Compounding Views	39.33	0.84	46.71X	0.84	46.71X
Entire Reflection Reconstruction	2108.40	79.16	26.63X	54.57	38.64X

“Multi-GPU Accelerated Refraction-Corrected Reflection Image Reconstruction for 3D Ultrasound Breast Imaging”, GTC 15

Overview of (NVIDIA) GPU Architecture

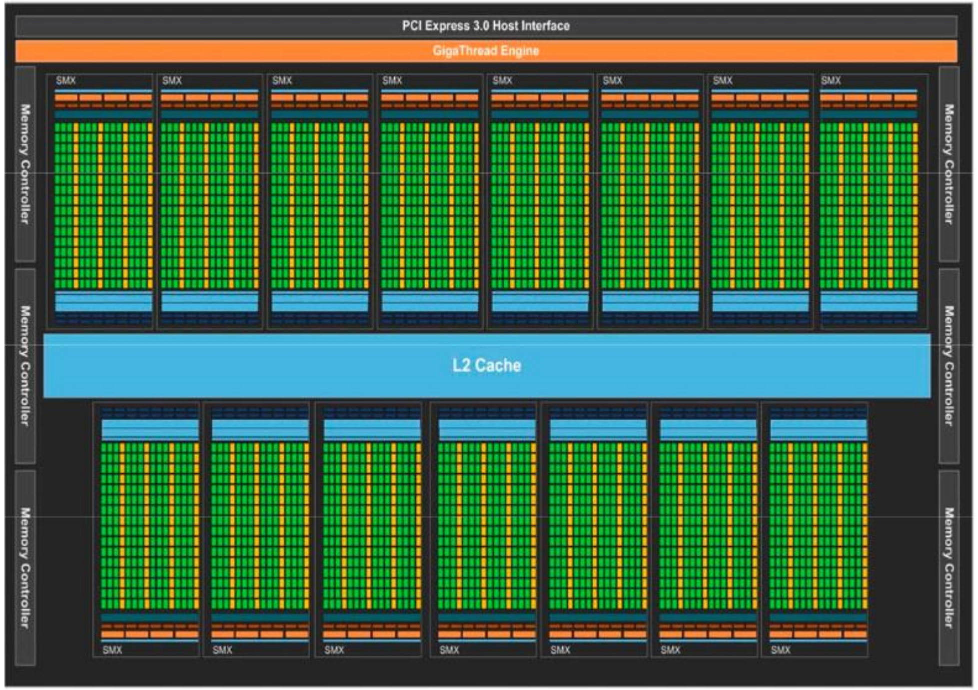
Architectural Generations

Discussion will focus on characteristics shared across architectural generations.

When important, differences between generations will be noted.

- Fermi ➔ Kepler ➔ Maxwell ➔ Pascal ➔ Volta

Much of the material in this section can be found in the white papers below.



http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf
https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf
https://developer.nvidia.com/maxwell-compute-architecture
http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
https://devblogs.nvidia.com/parallelforall/inside-volta/

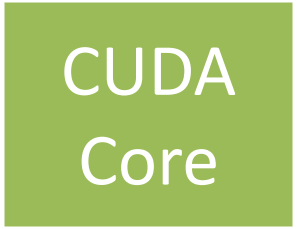
The CUDA Core

The CUDA Core is the finest granularity of execution on NVIDIA GPUs.

In a single cycle, executes either:

- 32-bit integer arithmetic instruction
- 32-bit floating point arithmetic instruction

CUDA core != x86 core.



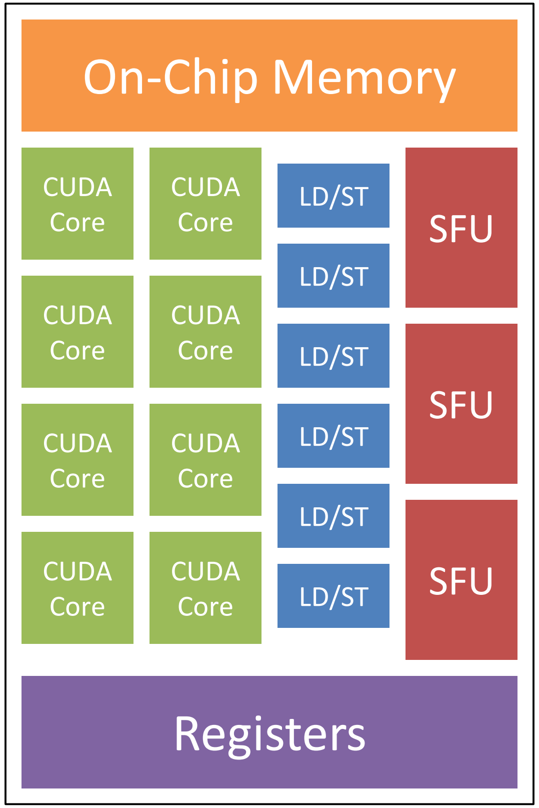
The Streaming Multiprocessor (SM)

SM is analogous to “core” in x86 processors

Consists of many CUDA cores

- 32 for Fermi, 192 for Kepler, 128 for Maxwell, 64 for Pascal
- All CUDA cores in the same SM execute in lock step
- Registers + on-chip memory are shared among threads running on this SM

Think “vector core” but with more flexibility.

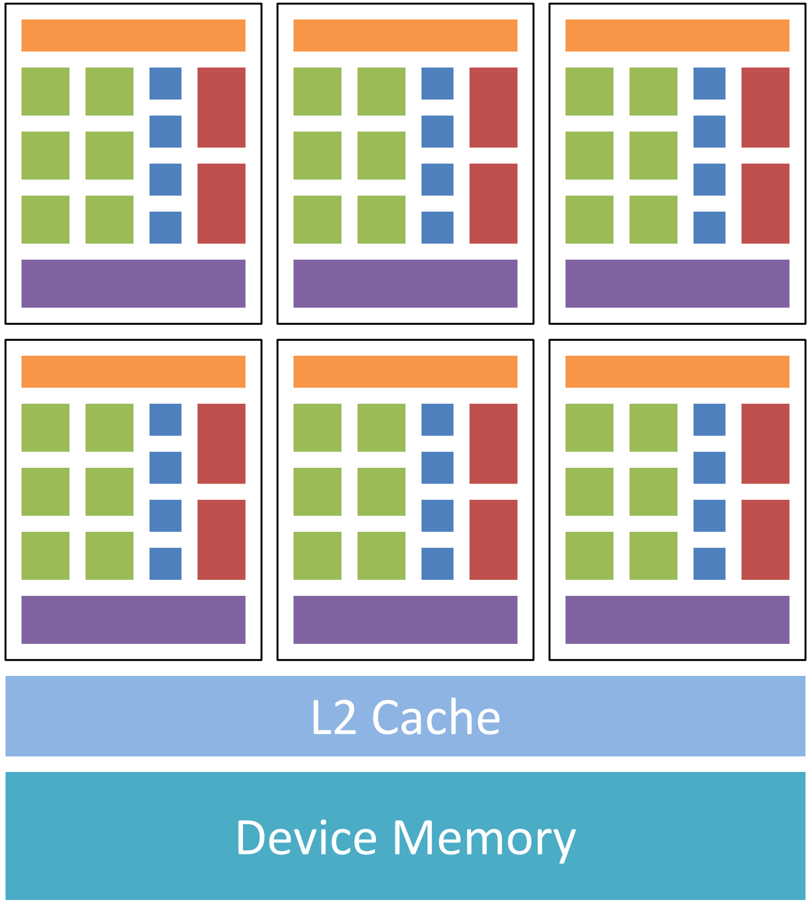


The GPU

Consists of O(10) SMs

L2 cache, device memory (DRAM) shared among all SMs

Off-chip latency ~100x worse than on-chip



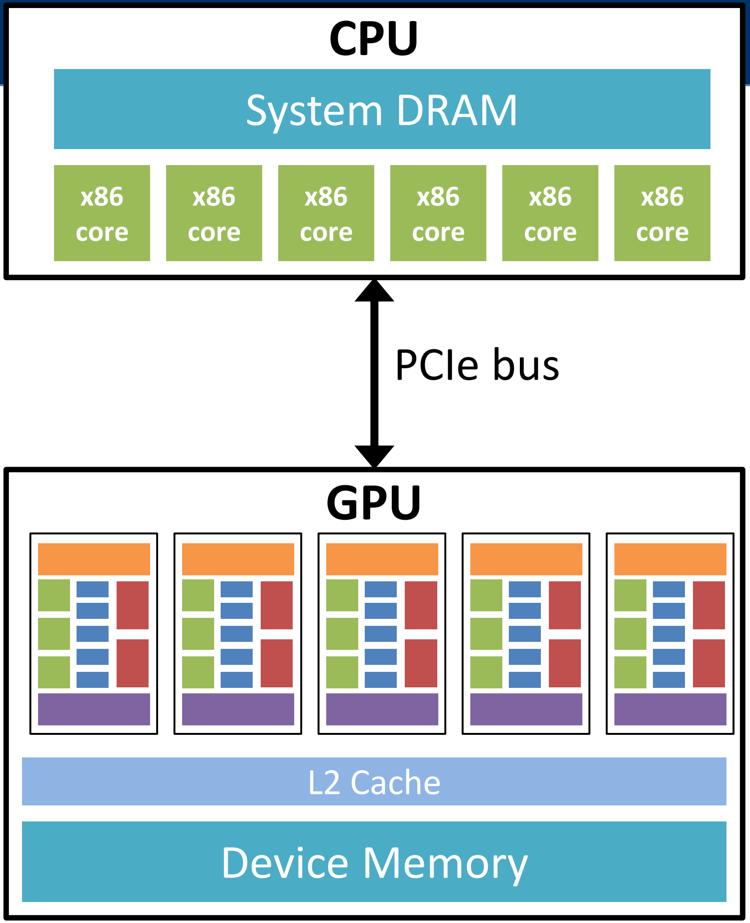
The Full System

GPUs are not standalone processors, require a CPU to manage them.

Commonly refer to CPU as "host" and GPU as "device".

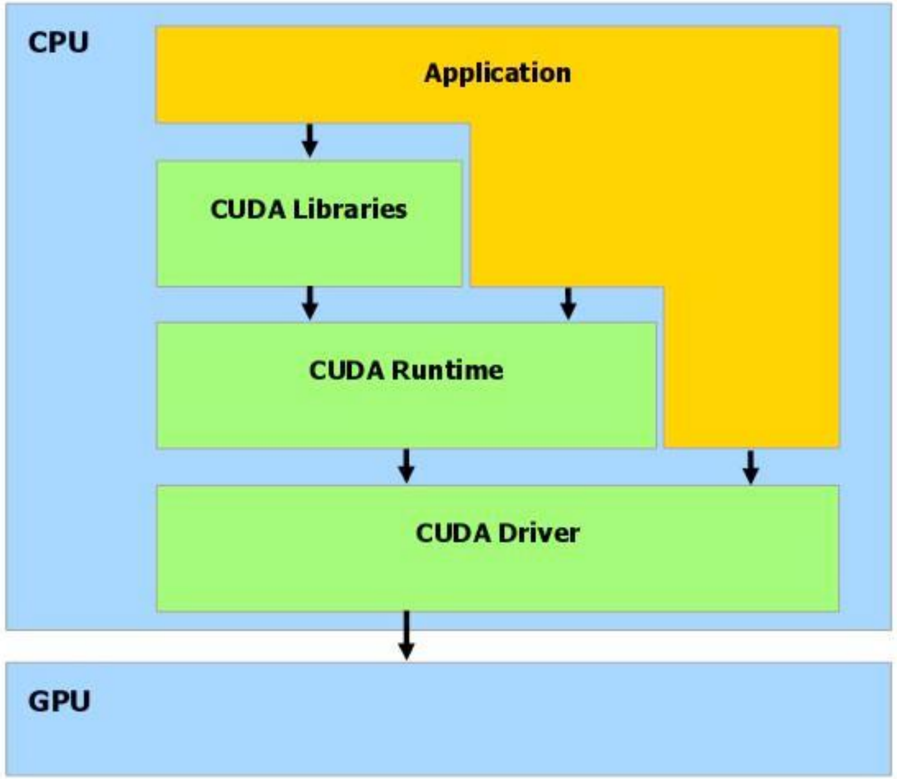
Connected by very high latency PCIe bus.

Analogous to two MPI processes, versus OpenMP/Cilk everything shared model.



Hands-On With CUDA, a Simple Example

CUDA Software Stack



http://www.comp.leeds.ac.uk/viznet/reports/GPU_report/GPUSTARReport_html.html

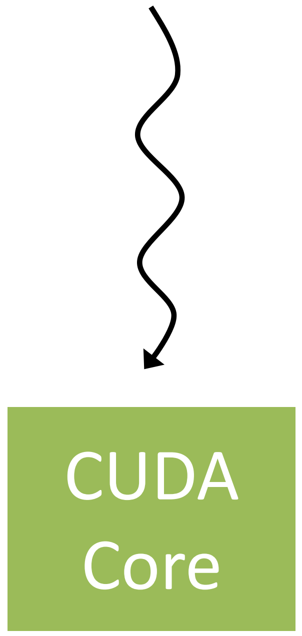
A Simple CUDA Example

Kernel is written as single-threaded code.

A single CUDA thread per CUDA core, each CUDA thread executes same kernel (similar to SPMD).

Let's start by considering vector addition: **`C = B + A`**

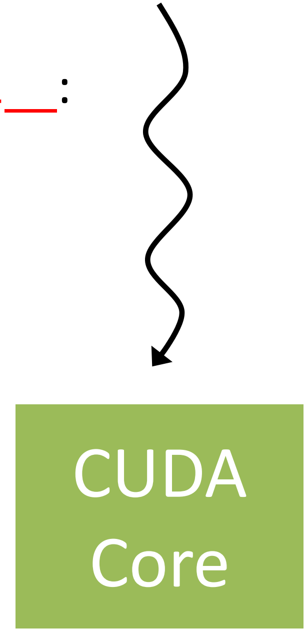
```
void vector_add(int *C, int *B, int *A, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```



A Simple CUDA Example

To enable GPU execution of this function, simply add __global__:

```
__global__ void vector_add(int *C, int *B, int *A,
    int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

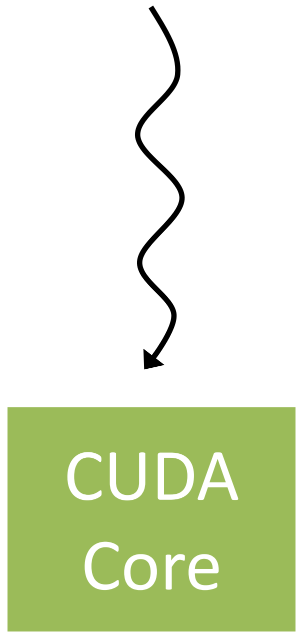


A Simple CUDA Example

CUDA function type qualifiers:

Qualifier	Executes on…	Callable from…
`__global__`	device	host
`__host__`	host	host
`__device__`	device	device

```
__device__ void bar(...) { ... }

__global__ void foo(...) {
  bar(...);
}
```

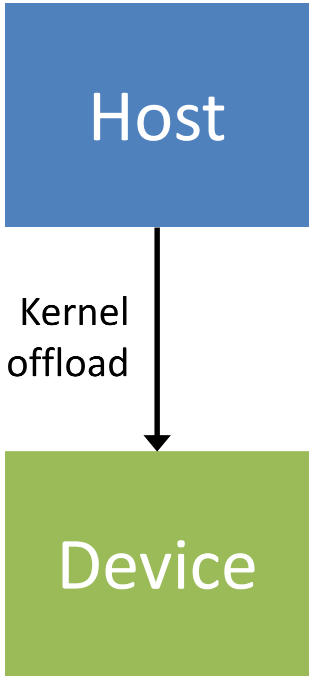


A Simple CUDA Example

Calling this kernel from the host requires special syntax to indicate that the function call runs on the device:

```
__global__ void vector_add(...) {
  ...
}

vector_add<<<1, 1>>>(C, A, B, N);
```

Run one thread on the GPU
(more on this later)



A Simple CUDA Example

One final step: the data.

Recall that host and device are physically discrete processors ➔ separate address spaces, a pointer on one has no meaning on the other.

Host

```
int *A = (int *)malloc(N * sizeof(int));
int *B = (int *)malloc(N * sizeof(int));
int *C = (int *)malloc(N * sizeof(int));

vector_add<<<1, 1>>>(C, A, B, N);
```

Device

```
__global__ void vector_add(int *C,
    int *A, int *B, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

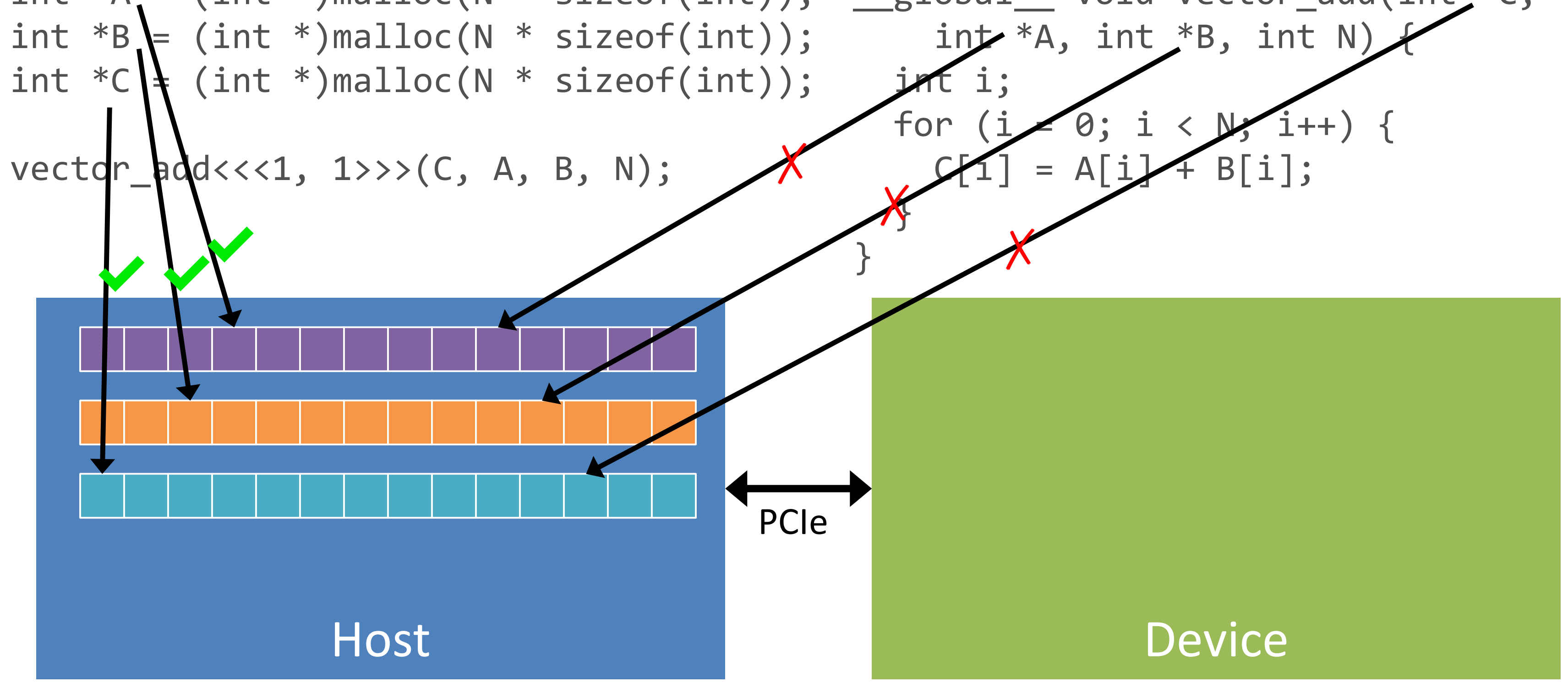
A Simple CUDA Example

Host

```
int *A = (int *)malloc(N * sizeof(int));
int *B = (int *)malloc(N * sizeof(int));
int *C = (int *)malloc(N * sizeof(int));

vector_add<<<1, 1>>>(C, A, B, N);
```

Device

```
__global__ void vector_add(int *C,
    int *A, int *B, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```



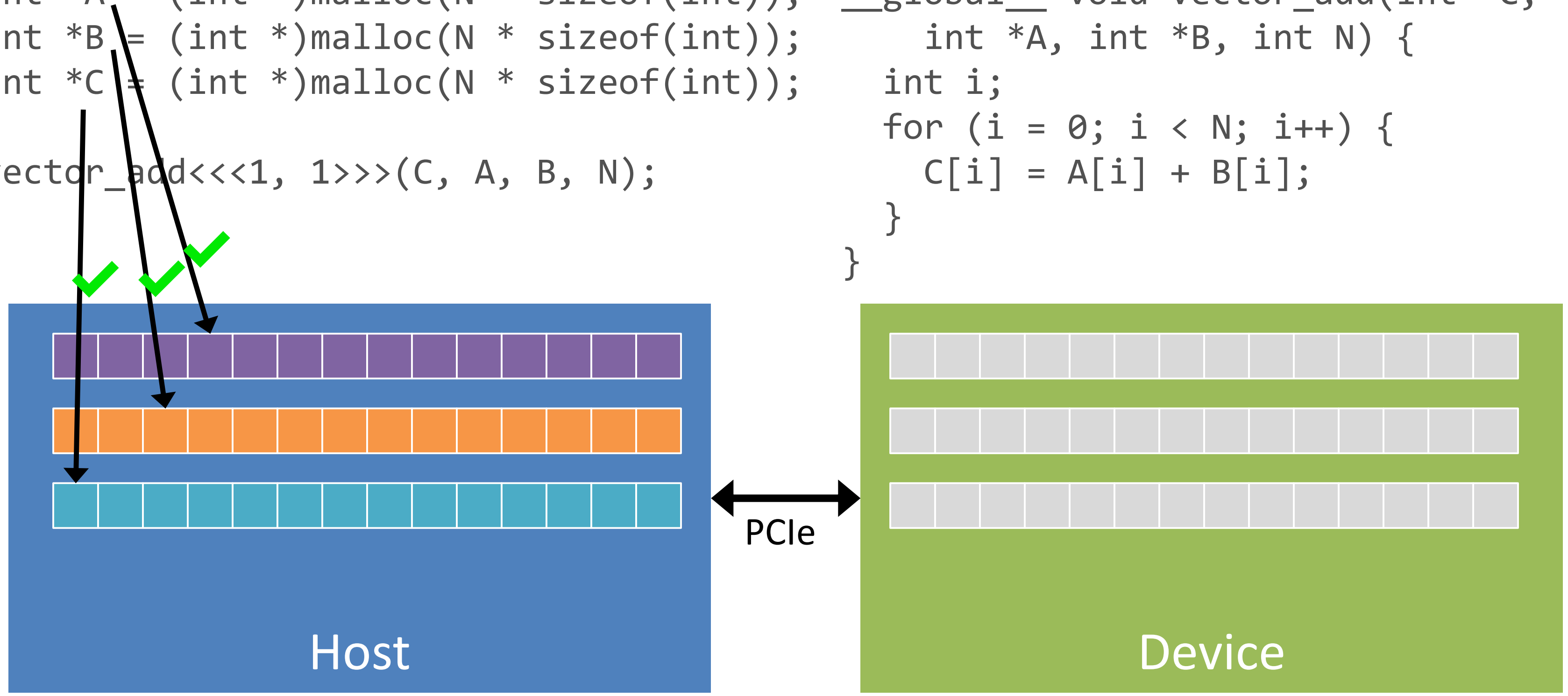
A Simple CUDA Example

Host

```
int *A = (int *)malloc(N * sizeof(int));
int *B = (int *)malloc(N * sizeof(int));
int *C = (int *)malloc(N * sizeof(int));

vector_add<<<1, 1>>>(C, A, B, N);
```

Device

```
__global__ void vector_add(int *C,
    int *A, int *B, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```



A Simple CUDA Example

Host

```
int *A = (int *)malloc(N * sizeof(int));
int *B = (int *)malloc(N * sizeof(int));
int *C = (int *)malloc(N * sizeof(int));

vector_add<<<1, 1>>>(C, A, B, N);
```

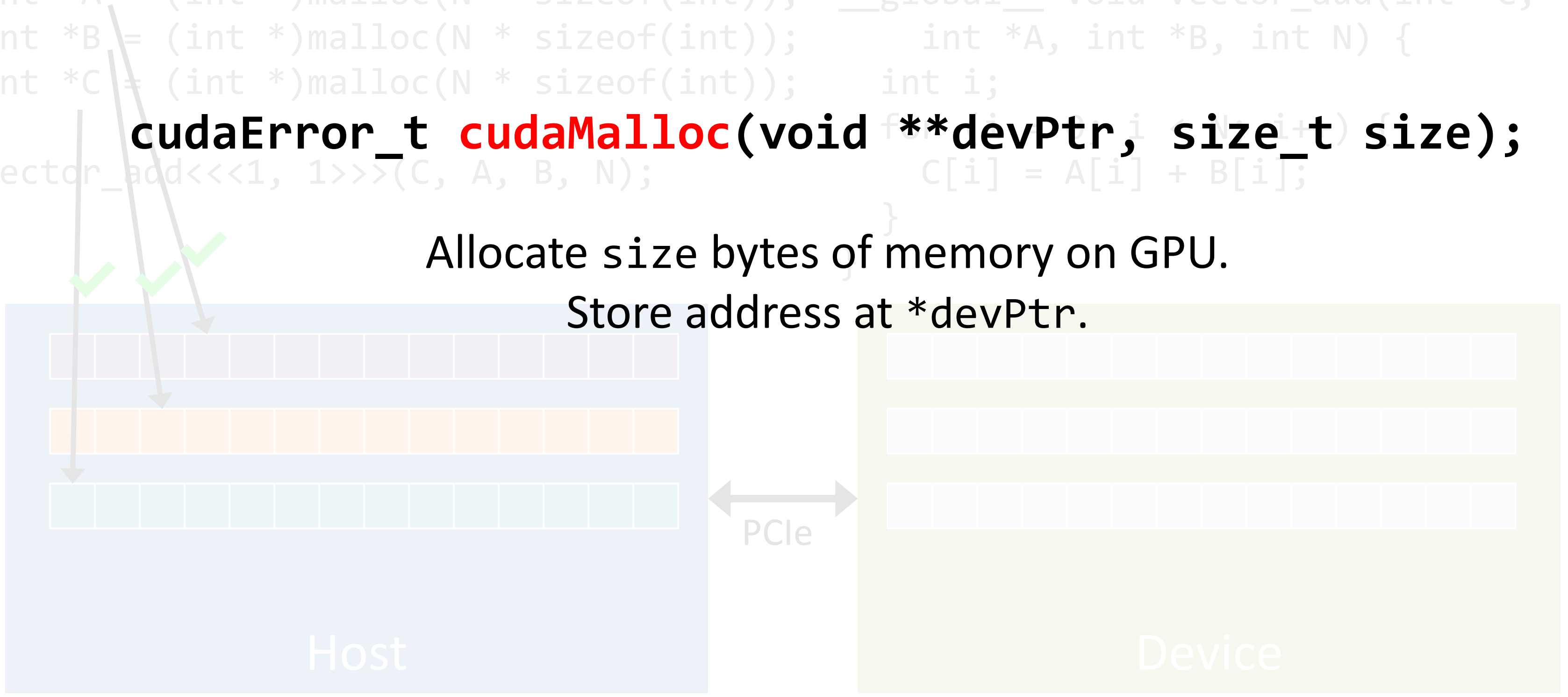
Device

```
__global__ void vector_add(int *C,
    int *A, int *B, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```
cudaError_t cudaMalloc(void **devPtr, size_t size);
```

Allocate `size` bytes of memory on GPU.

Store address at `*devPtr`.



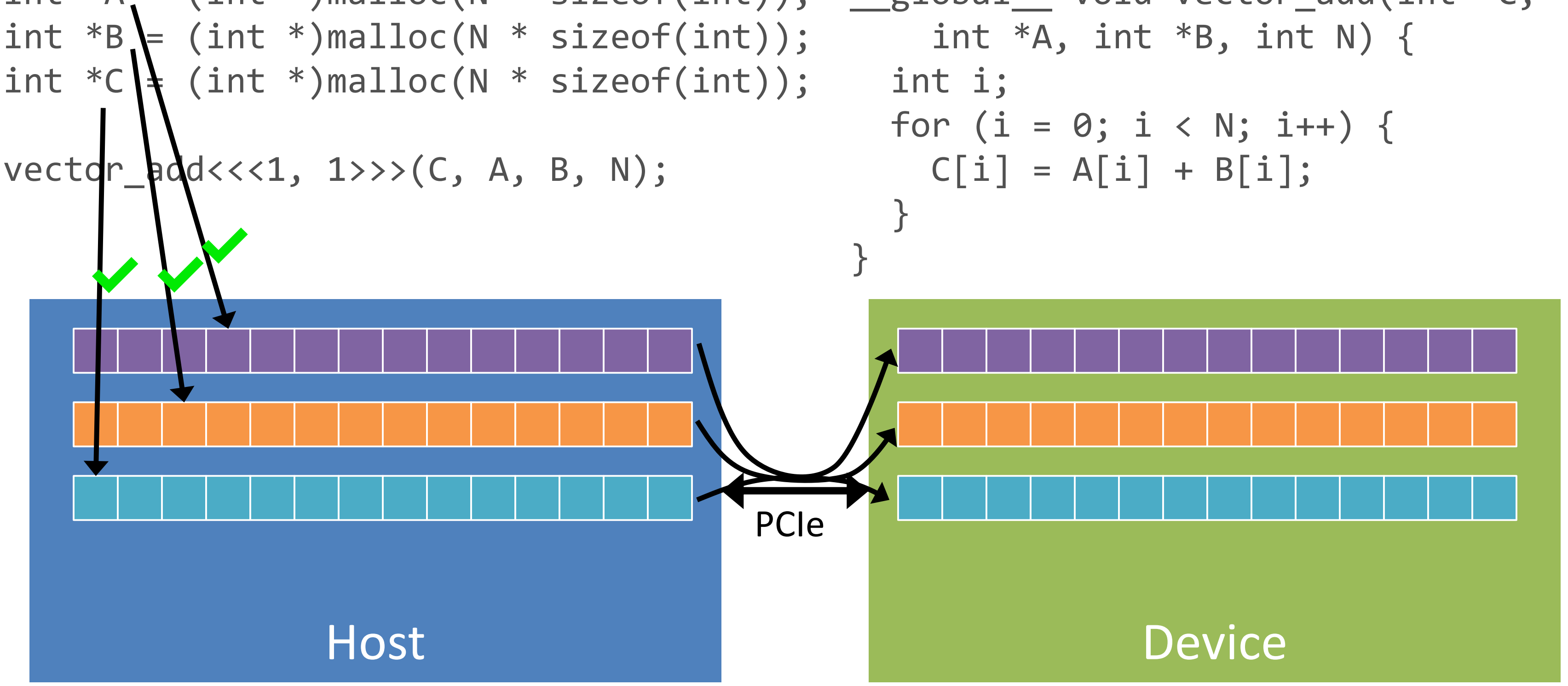
A Simple CUDA Example

Host

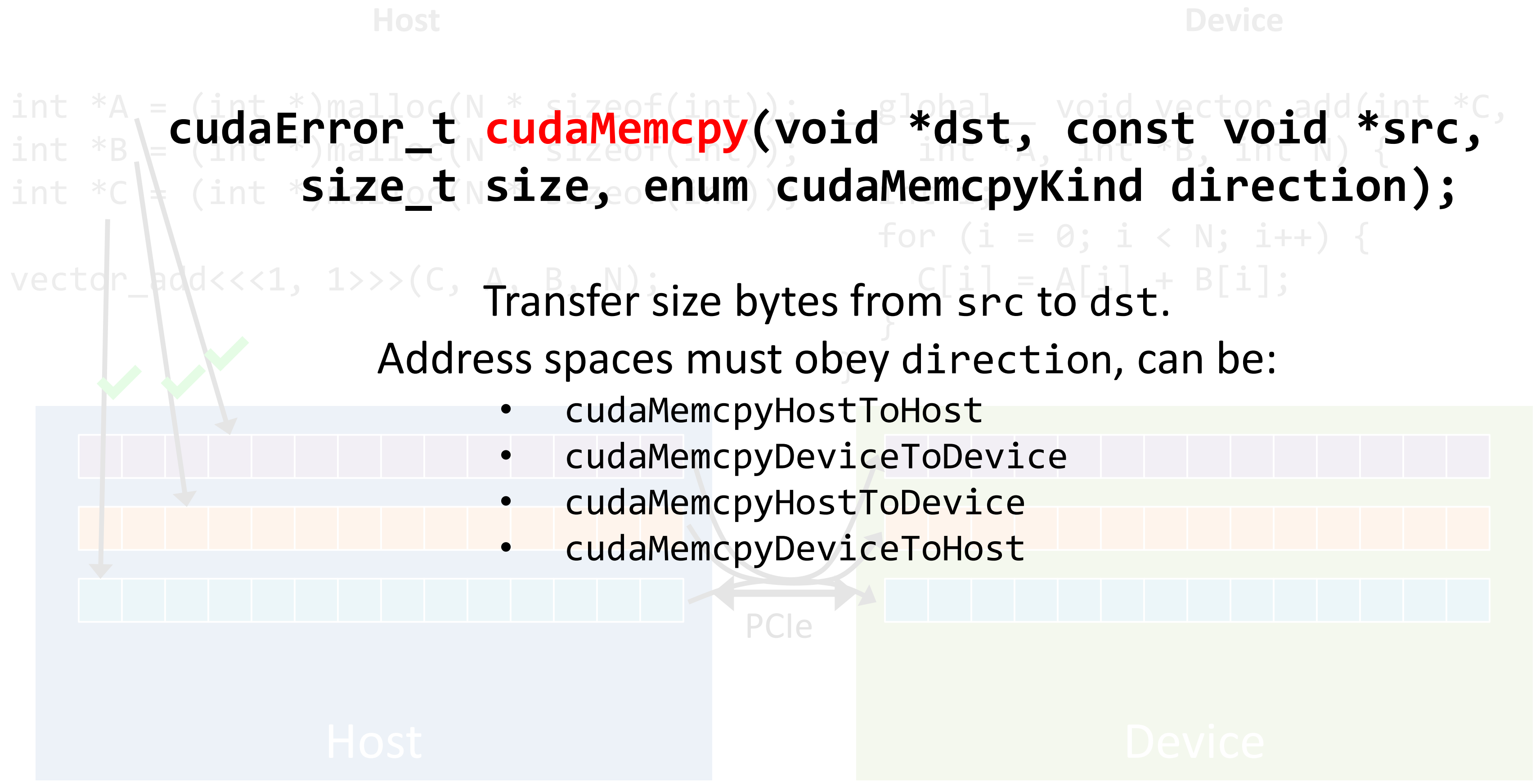
```
int *A = (int *)malloc(N * sizeof(int));
int *B = (int *)malloc(N * sizeof(int));
int *C = (int *)malloc(N * sizeof(int));

vector_add<<<1, 1>>>(C, A, B, N);
```

Device

```
__global__ void vector_add(int *C,
    int *A, int *B, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```



A Simple CUDA Example



```
cudaError_t cudaMemcpy(void *dst, const void *src,
                       size_t size, enum cudaMemcpyKind direction);
```

Transfer size bytes from src to dst.

Address spaces must obey direction, can be:

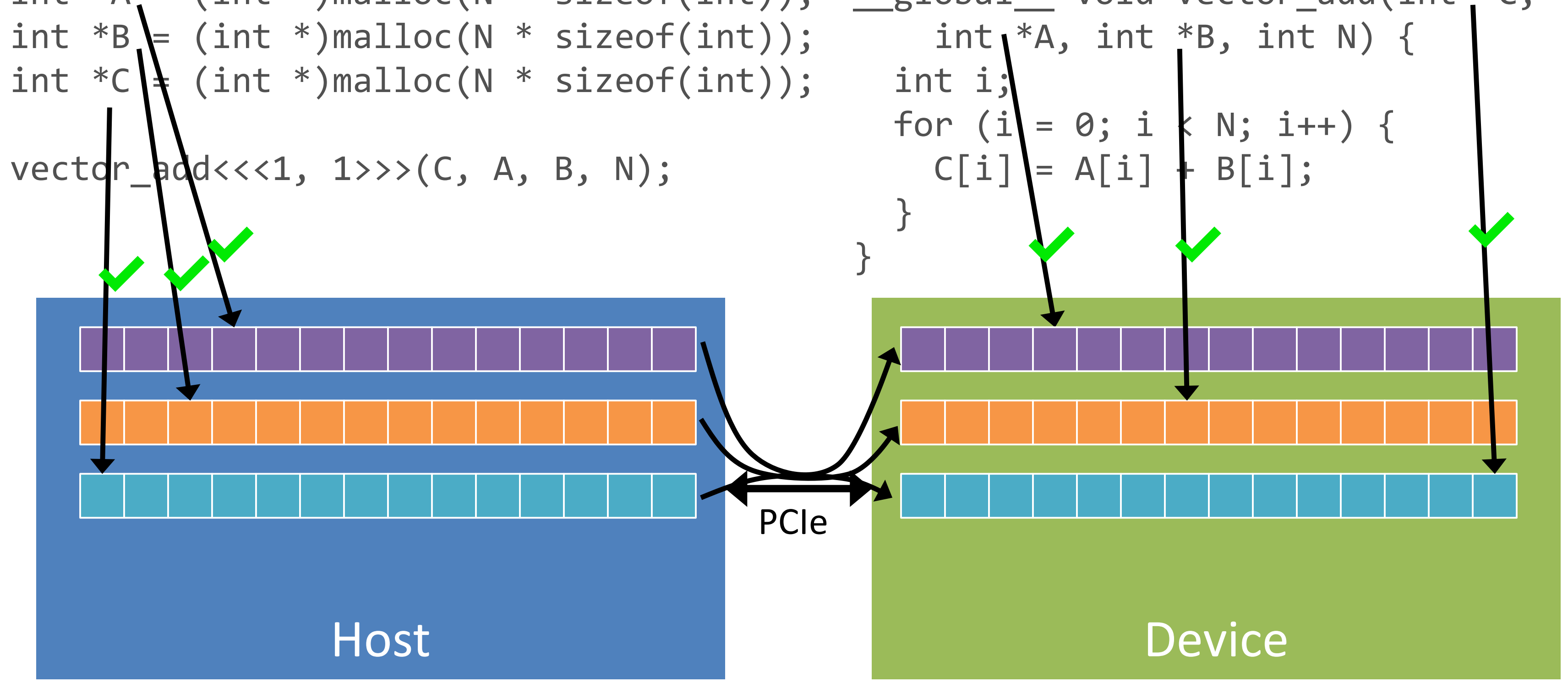
- cudaMemcpyHostToHost
- cudaMemcpyDeviceToDevice
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost

A Simple CUDA Example

Host **Device**

```
int *A = (int *)malloc(N * sizeof(int));
int *B = (int *)malloc(N * sizeof(int));
int *C = (int *)malloc(N * sizeof(int));

vector_add<<<1, 1>>>(C, A, B, N);
```

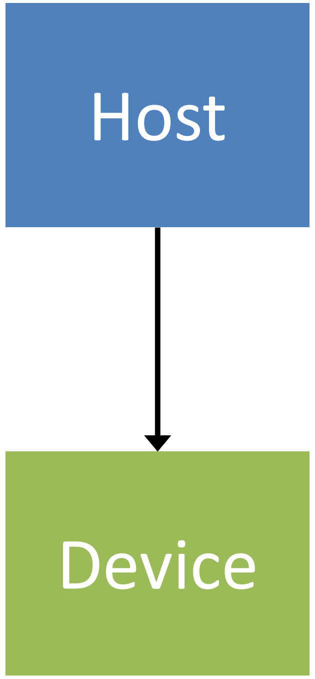
```
__global__ void vector_add(int *C,
    int *A, int *B, int N) {
  int i;
  for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```



A Simple CUDA Example

```
int *A, *d_A;
A = (int *)malloc(N * sizeof(int));
cudaMalloc((void **)&d_A, N * sizeof(int));
cudaMemcpy(d_A, A, N * sizeof(int), cudaMemcpyHostToDevice);

...

vector_add<<<1, 1>>>(d_C, d_A, d_B, N);
```

```
cudaError_t cudaMalloc(void **devPtr, size_t size);

cudaError_t cudaMemcpy(void *dst, const void *src,
    size_t size, enum cudaMemcpyKind direction);

cudaError_t cudaFree(void *addr);
```



A Simple CUDA Example

Building CUDA programs requires **nvcc**. Supports all of the usual flags you expect from a compiler.

```
$ cd ~/bootcamp-gpu/src/00_vecadd
$ nvcc -arch=sm_20 vecadd.cu -o vecadd
```

↑	↑	↑	↑
NVIDIA's GPU compiler	Compile and optimize for the Fermi architecture	Input CUDA file	Output executable

*Makefiles will be provided for your convenience

Hands On - 00_vecadd

Don’t forget to follow the instructions on slide 1.

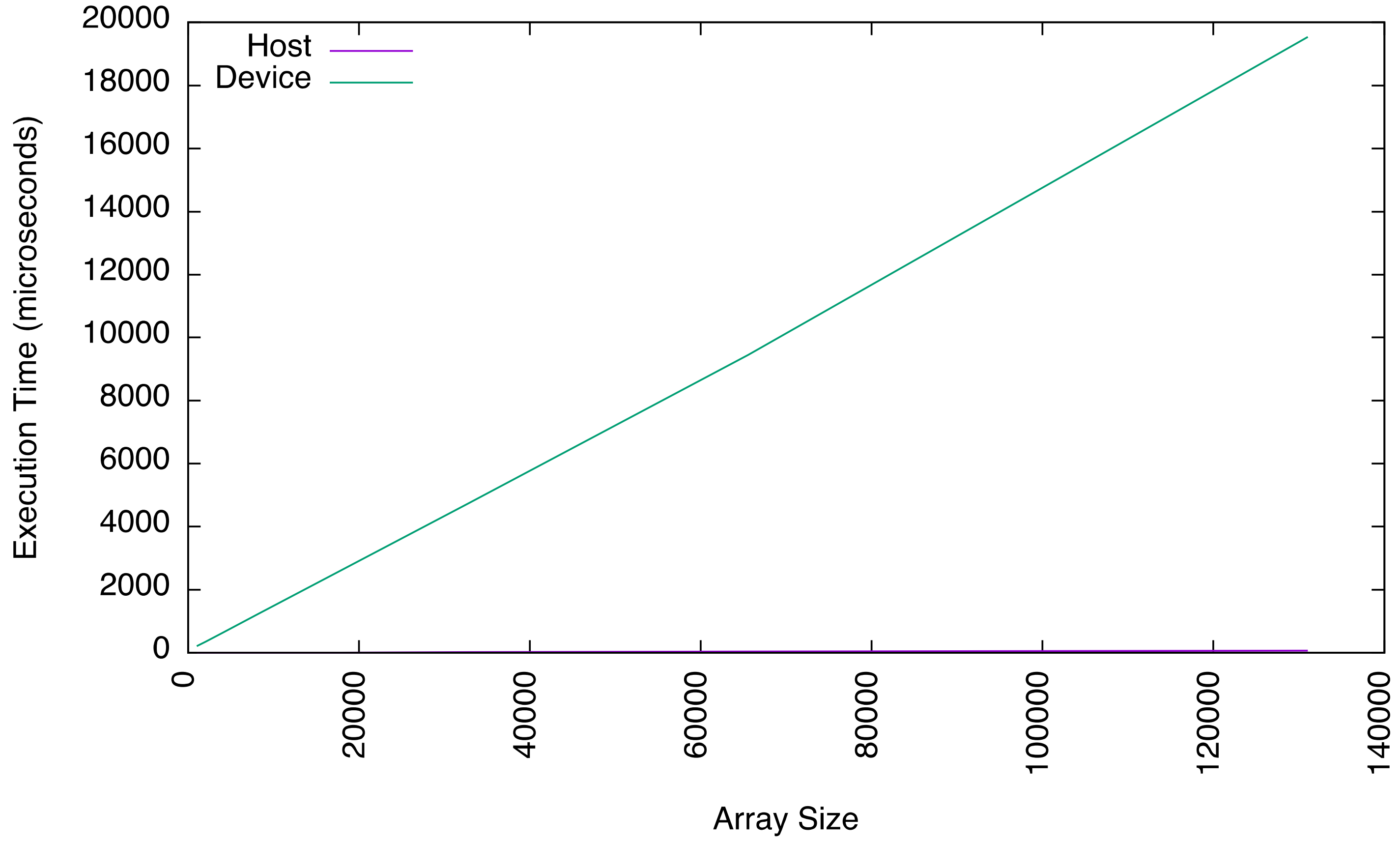
```
$ cd ~/bootcamp-gpu/src/00_vecadd
$ make
$ vim/emacs vecadd.cu # Get familiar with the code
$ ./vecadd 131072 # Vectors with 131,072 elements
```

What kind of performance do you observe?

Does it change relatively with different input sizes?

```
$ git clone git@github.com:agrippa/hpc-bootcamp.git $HOME/bootcamp-gpu
```

Hands On - 00_vecadd



Review - Basic CUDA

Running a function on the GPU requires 1) `__global__` annotation, and 2) kernel launch syntax (`<<<...>>>`).

CUDA forces you to think about discrete address spaces and manage coherency.

```
cudaError_t cudaMalloc(void **devPtr, size_t size);

cudaError_t cudaMemcpy(void *dst, const void *src,
    size_t size, enum cudaMemcpyKind direction);

cudaError_t cudaFree(void *addr);
```

Copying and pasting code is not the path to performance in CUDA.

nvprof

nvprof is a command-line profiler for CUDA applications.

```
$ nvprof --help
Usage: nvprof [options] [CUDA-application] [application-arguments]
...
```

Can be used in a number of modes:

- Default/Summary Mode
- API Trace Mode
- Event/Metric Summary Mode
- Event/Metric Trace Mode

http://docs.nvidia.com/cuda/profiler-users-guide/

Hands On – nvprof Summary mode

A good starting point is usually Default/Summary mode.

```
$ nvprof ./vecadd 131072
```

http://docs.nvidia.com/cuda/profiler-users-guide/

Hands On – nvprof Summary mode

A good starting point is usually Default/Summary mode.

```
$ nvprof ./vecadd 131072
```

```
==17036== NVPROF is profiling process 17036, command: ./vecadd 131072
Finished! All 131072 elements validate.
Took 621 microseconds on the host
Took 19725 microseconds on the device, 0.03148x speedup
==17036== Profiling application: ./vecadd 131072
==17036== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.61%  18.656ms         1  18.656ms  18.656ms  18.656ms  vector_add(int*, int*, int*, int)
  0.95%  180.03us         2  90.015us  89.951us  90.079us  [CUDA memcpy HtoD]
  0.44%  82.655us         1  82.655us  82.655us  82.655us  [CUDA memcpy DtoH]
```

http://docs.nvidia.com/cuda/profiler-users-guide/

nvprof

Diving deeper requires Event/Metric Summary Mode.

```
$ nvprof --query-events # List all enabled events
$ nvprof --query-metrics # List all enabled metrics
```

Events are raw data collected by the hardware, e.g. # instructions (`inst_issued`)

Metrics are derived based on events, e.g. # instructions per cycle (`issued_ipc`)

Hands On – nvprof Metrics

Diving deeper requires Event/Metric Summary Mode.

1. `ipc`: # of instructions executed per cycle on the GPU
2. `sm_efficiency`: % of time an SM on the GPU is doing useful work
3. `alu_fu_utilization`: indicates how well the CUDA cores are being used

```
$ nvprof --metrics ipc,sm_efficiency,alu_fu_utilization ./vecadd 131072
```

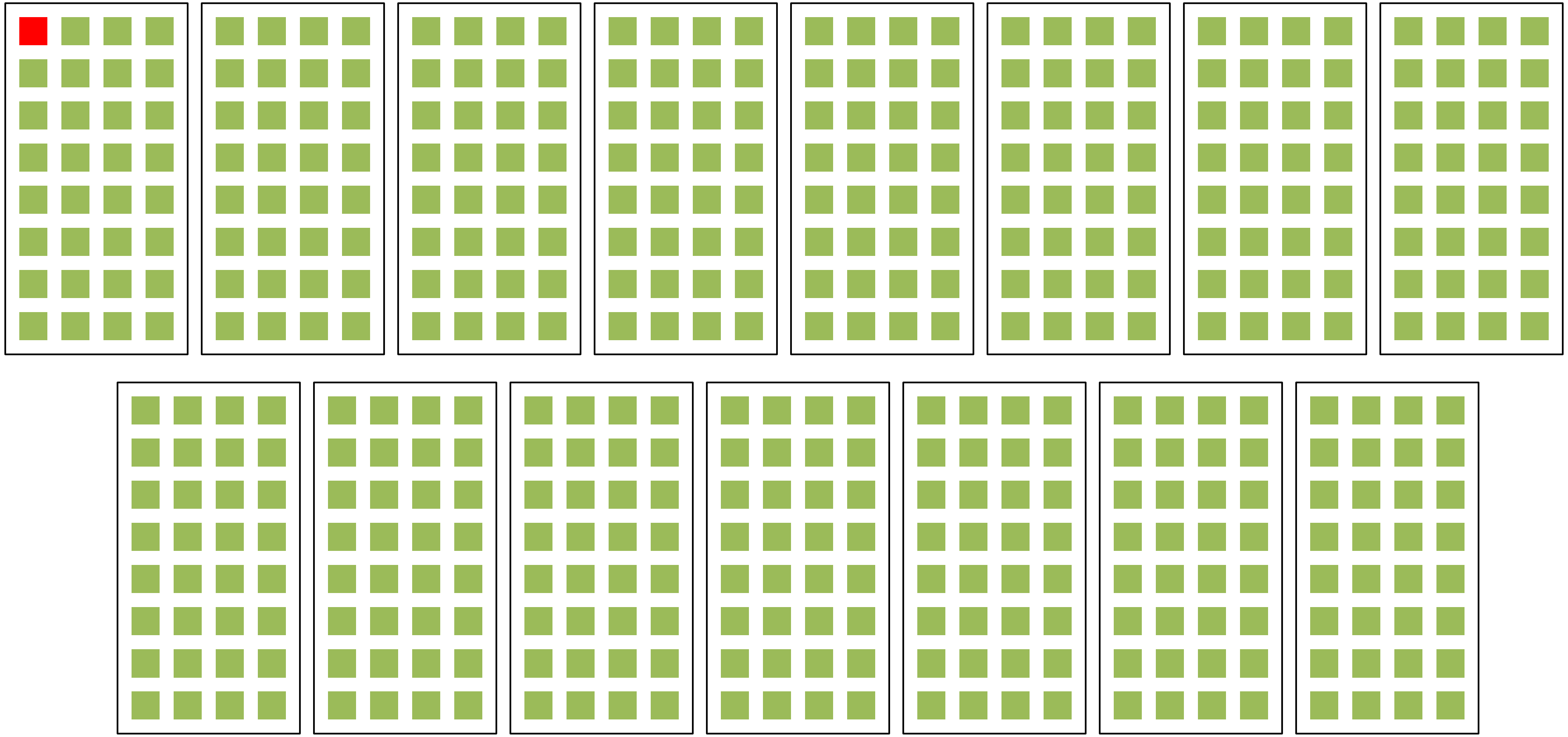
Hands On – nvprof Metrics

Diving deeper requires Event/Metric Summary Mode.

```
$ nvprof --metrics ipc,sm_efficiency,alu_fu_utilization ./vecadd 131072
```

```
==30413== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Finished! All 131072 elements validate.
Took 621 microseconds on the host
Took 174666 microseconds on the device, 0.00356x speedup
==30413== Profiling application: ./vecadd 131072
==30413== Profiling result:
==30413== Metric result:
Invocations                          Metric Name                        Metric Description         Min ...
Device "Tesla M2050 (0)"
        Kernel: vector_add(int*, int*, int*, int)
          1                        sm_efficiency                   Multiprocessor Activity       7.14% ...
          1                                  ipc                              Executed IPC    0.103982 ...
          1                   alu_fu_utilization      Arithmetic Function Unit Utilization     Low (1) ...
```

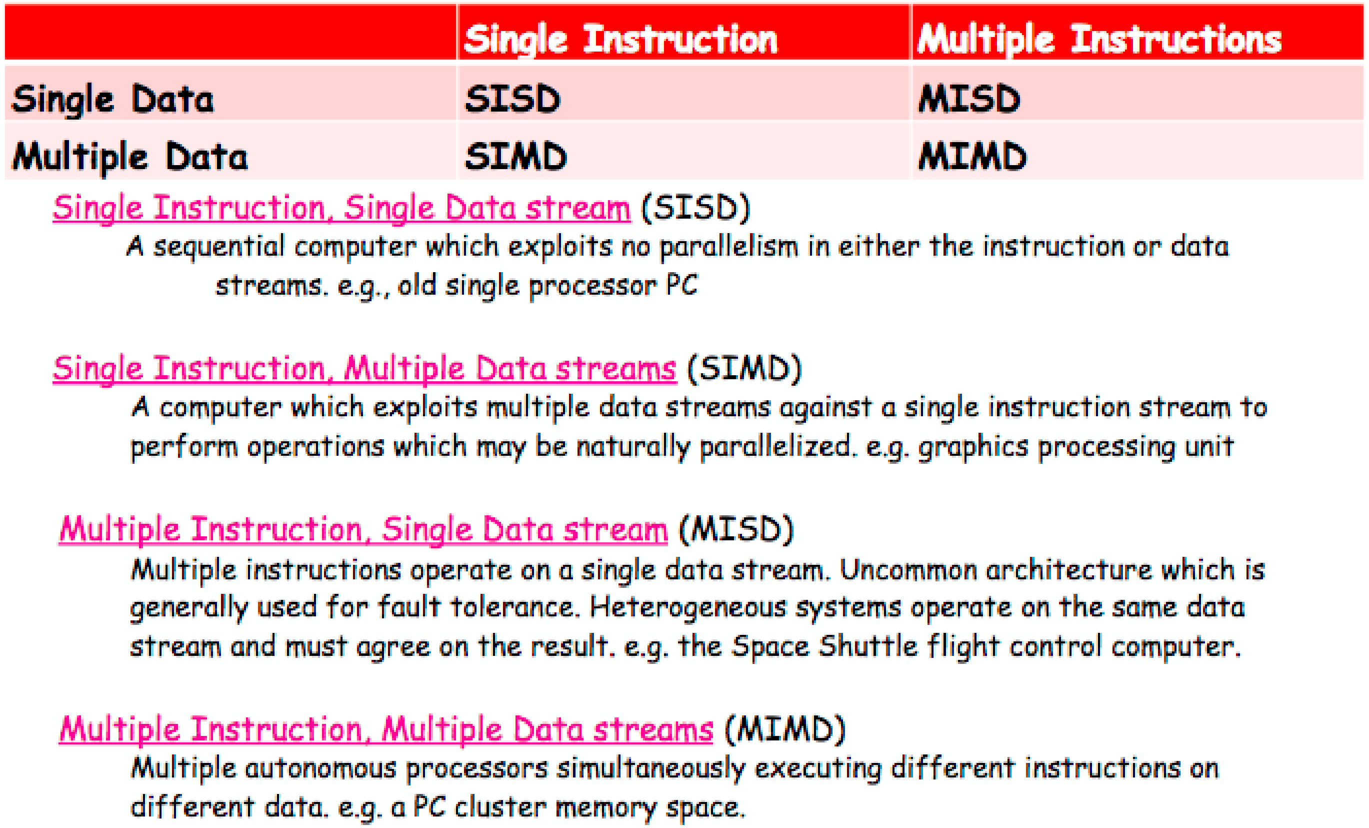
nvprof



CUDA Execution Model

Or How to Write Parallel CUDA Programs

Flynn’s Taxonomy



	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Single Instruction, Single Data stream (SISD)

A sequential computer which exploits no parallelism in either the instruction or data streams. e.g., old single processor PC

Single Instruction, Multiple Data streams (SIMD)

A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. e.g. graphics processing unit

Multiple Instruction, Single Data stream (MISD)

Multiple instructions operate on a single data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. e.g. the Space Shuttle flight control computer.

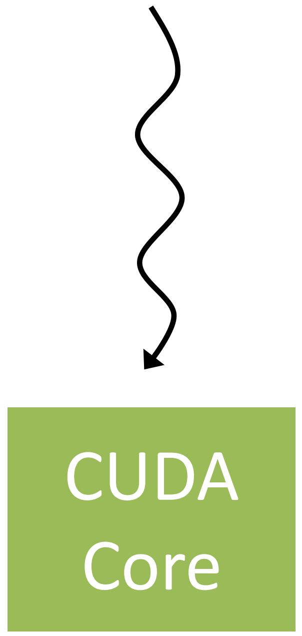
Multiple Instruction, Multiple Data streams (MIMD)

Multiple autonomous processors simultaneously executing different instructions on different data. e.g. a PC cluster memory space.

CUDA Execution Model

Recall:

- Programmer writes single-threaded kernel
- 1 **CUDA thread** per CUDA core.



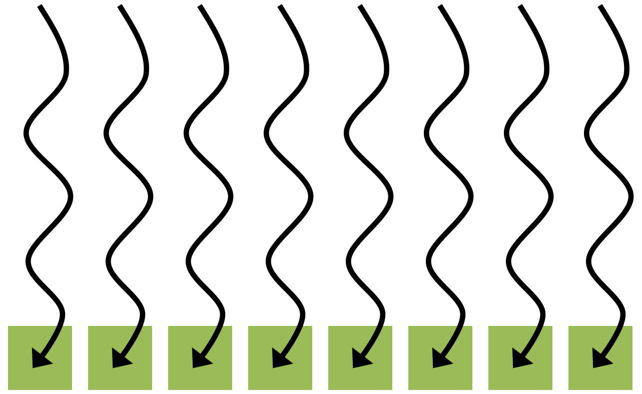
CUDA Execution Model

Recall:

- Programmer writes single-threaded kernel
- 1 **CUDA thread** per CUDA core.

Threads are scheduled in groups of 32, called a **warp**.

- Warps execute in lock step (same instruction at same cycle)
- All threads in warp on same SM



CUDA Execution Model

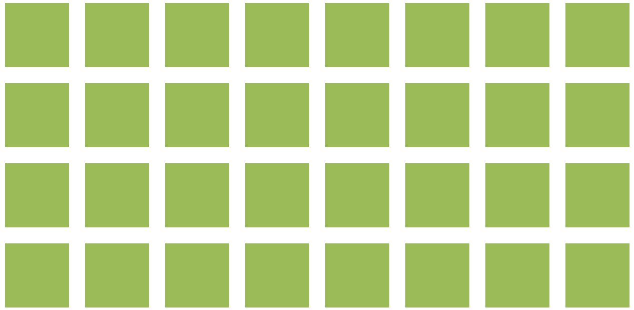
Recall:

- Programmer writes single-threaded kernel
- 1 **CUDA thread** per CUDA core.

Threads are scheduled in groups of 32, called a **warp**.

Warps are grouped into **thread blocks**.

- Warps in a block execute on same SM
- Warp execution in same thread block can be interleaved with each other on SM
- SM resources are partitioned among all threads in resident thread blocks



e.g. 4 warps in one thread block

CUDA Execution Model

Recall:

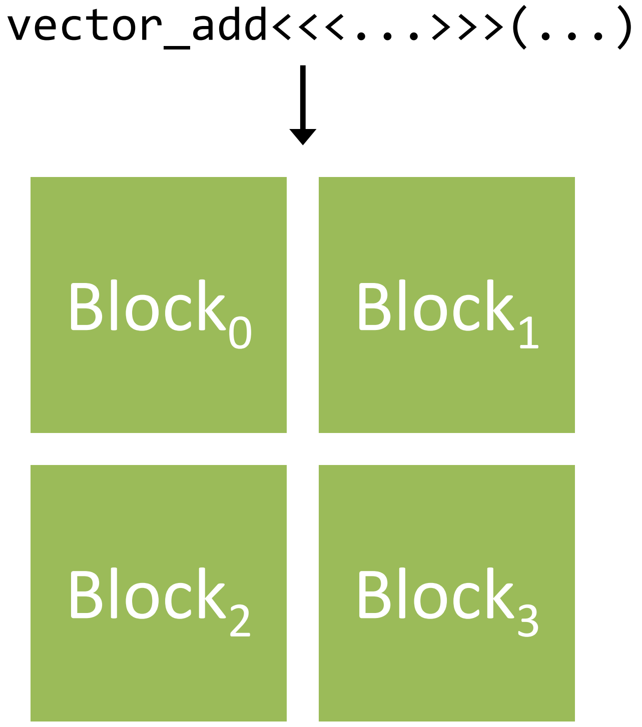
- Programmer writes single-threaded kernel
- 1 **CUDA thread** per CUDA core.

Threads are scheduled in groups of 32, called a **warp**.

Warps are grouped into **thread blocks**.

A single kernel **grid** can include many thread blocks.

- Kernel call is not complete until all thread blocks within it complete.



CUDA Execution Model

Blocks and grids are configurable by kernel launch arguments:

```
        vector_add<<<nblocks, nthreads_per_block>>>(...)
```

```
vector_add<<<1, 1>>>(...); // 1 block with 1 thread inside it

vector_add<<<1, 32>>>(...); // 1 block with 1 full warp

vector_add<<<1, 256>>>(...); // 1 block with 256 threads (8 warps)

vector_add<<<N, 256>>>(...); // N blocks with 256 threads/block
```

CUDA Execution Model

Blocks and grids can also be multi-dimensional:

```
// Equivalent to vector_add<<<1, 256>>>(...);
dim3 1d_block(256, 1, 1);
vector_add<<<1, 1d_block>>>(...);

dim3 block(16, 16); // 2D 16x16 block of threads, 256 total

dim3 grid(4, 4); // 2D 4x4 grid of thread blocks, 16 total

kernel<<<grid, block>>>(...);
```

CUDA Execution Model

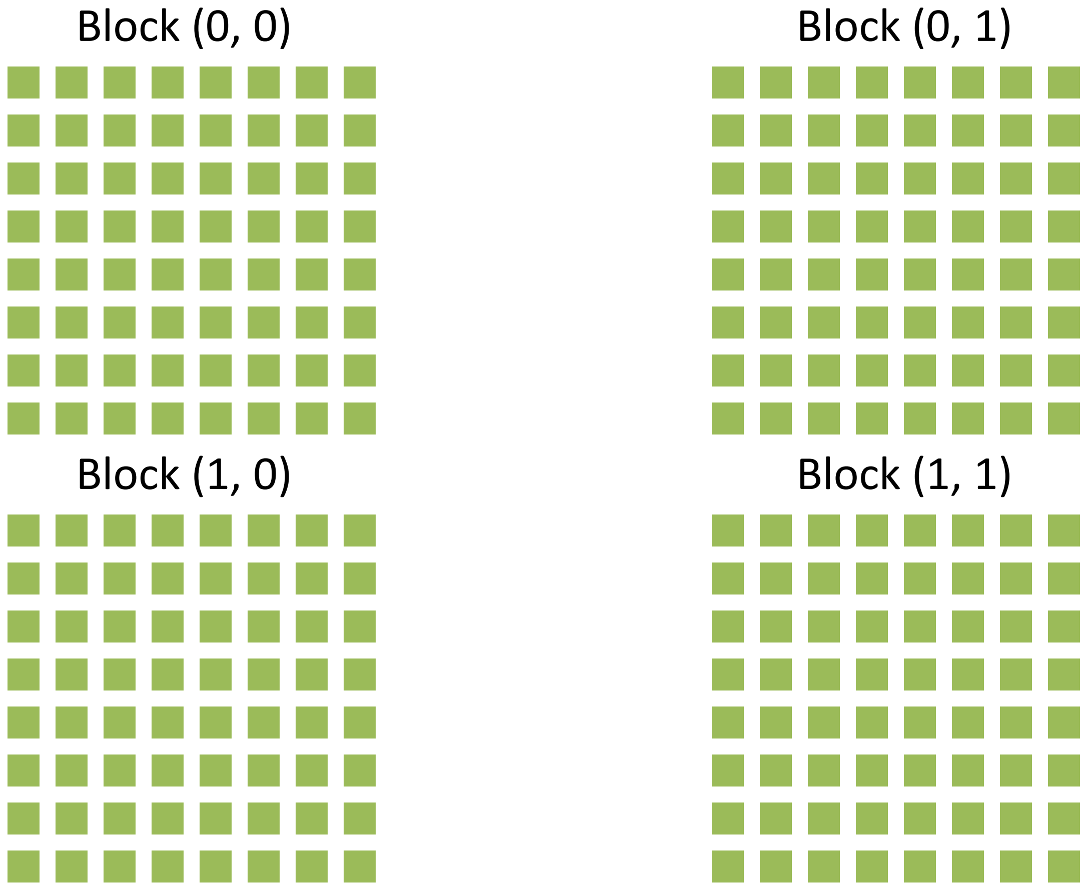
Special CUDA kernel variables can be used to check the coordinates of a thread in an executing grid.

Variable Name	Description
`threadIdx.x, threadIdx.y, …`	Offset of a thread within a thread block
`blockIdx.x, blockIdx.y, …`	Offset of a block within a grid
`blockDim.x, …`	# of threads per block
`gridDim.x, …`	# of blocks in grid
`blockIdx.x * blockDim.x + threadIdx.x`	One way to calculate globally unique thread ID for 1D blocks and grids

CUDA Execution Model

```
__global__ void matrix_add(...) {
  const int row = blockIdx.x * blockDim.x + threadIdx.x;
  const int col = blockIdx.y * blockDim.y + threadIdx.y;
  ...
}

dim3 block(8, 8); // 2D 8x8 block of threads, 64 total
dim3 grid(2, 2); // 2D 2x2 grid of thread blocks, 4 total

vector_add<<<grid, block>>>(...);
```

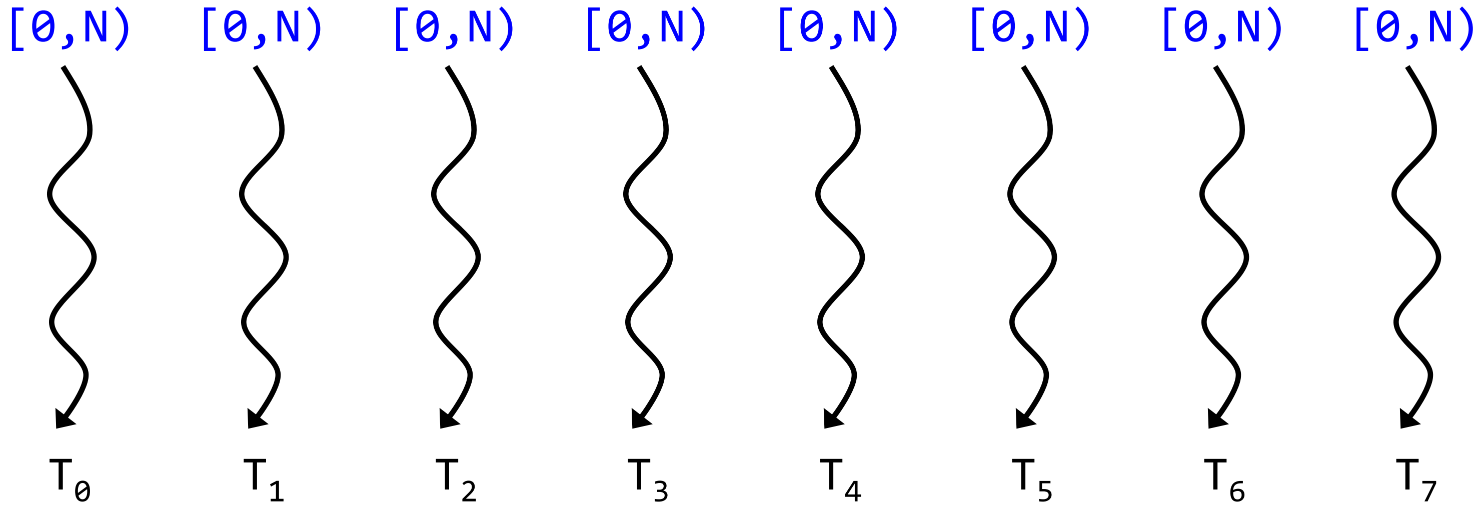
CUDA Execution Model



CUDA Execution Model

Launch configuration alone is insufficient for our `vector_add` to run in parallel.

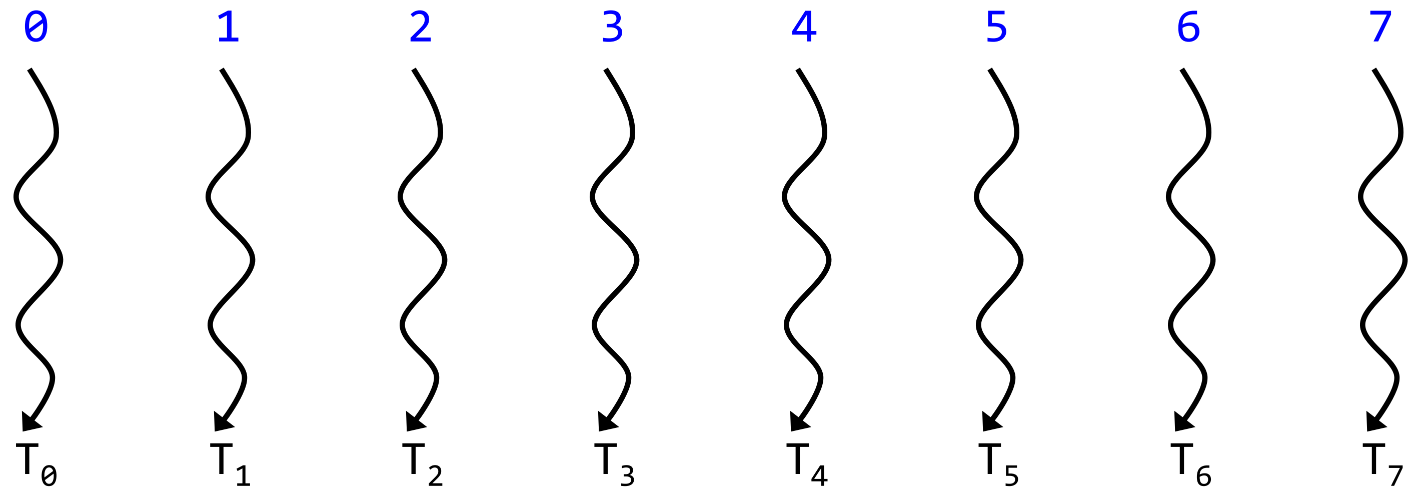
```
vector_add<<<N / 256, 256>>>(C, A, B, N);
```



CUDA Execution Model

Must also change the work performed by the kernel.

```
__global__ void vector_add(int *C, int *B, int *A, int N) {
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  C[i] = A[i] + B[i];
}
```



Hands On – Adding CUDA Parallelism

Using the previous slides as a guide, add parallelism to the vecadd example:

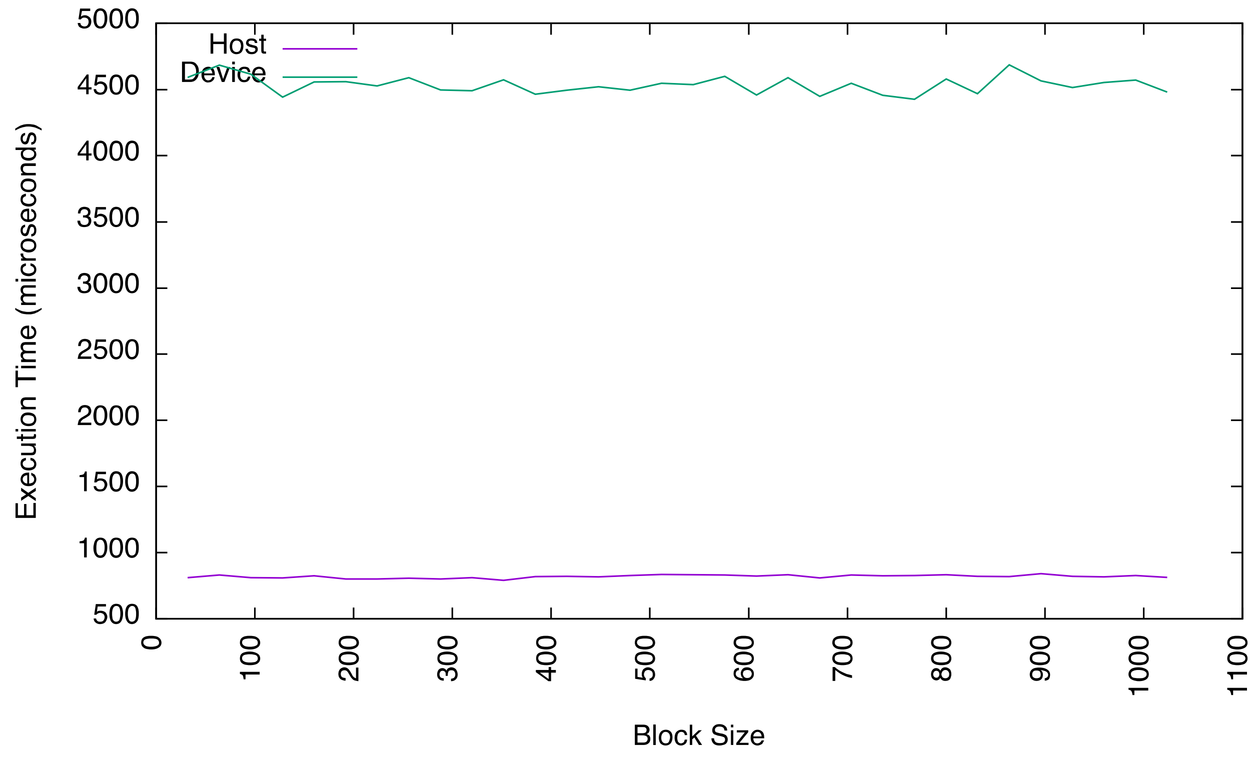
1. Change the <<<...>>> launch configuration to use B threads per block, N/B blocks
   - Be careful that the N you run is evenly divisible by B
   - CUDA supports up to 1,024 threads per block
2. Change the kernel to process a single data point

Does performance improve?

```
$ ./vecadd 1048576
```

Does block size affect performance?

Hands On – Adding CUDA Parallelism



More similar host and device performance (was ~150,000 us).

No clear trends in performance as a function of block size (yet).

Still not great performance, time to go back to `nvprof`.

Hands On – Adding CUDA Parallelism

`$HOME/bootcamp-gpu/src/01_vecadd` contains an example solution that allows configuring threads per block from the command line:

```
$ cd $HOME/bootcamp-gpu/src/01_vecadd
$ ./vecadd <N> <threads-per-block>
```

Try:

```
$ nvprof --metrics ipc,sm_efficiency,alu_fu_utilization \
      ./vecadd 1048576 256

$ nvprof ./vecadd 1048576 256
```

Hands On – Adding CUDA Parallelism

```
$ nvprof --metrics ipc,sm_efficiency,alu_fu_utilization ./vecadd 1048576
```

```
==32216== NVPROF is profiling process 32216, command: ./vecadd 1048576
==32216== Warning: Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Finished! All 1048576 elements validate using 256 threads per block.
Took 5032 microseconds on the host
Took 21110 microseconds on the device, 0.23837x speedup
==32216== Profiling application: ./vecadd 1048576
==32216== Profiling result:
==32216== Metric result:
Invocations                          Metric Name                    Metric Description        Min
Device "Tesla M2050 (0)"
        Kernel: vector_add(int*, int*, int*, int)
          1                        sm_efficiency               Multiprocessor Activity     96.20% …
          1                                  ipc                          Executed IPC   0.657917 …
          1                   alu_fu_utilization  Arithmetic Function Unit Utilization    Low (3) …
```

0.103982

7.14 %

Low (1)

Hands On – Adding CUDA Parallelism

```
$ nvprof ./vecadd 1048576 256
```

```
[jmg3@gpu-014 01_vecadd]$ /opt/apps/software/Compiler/GCC/4.4.7/CUDA/6.5.14/bin/nvprof  ./vecadd
1048576 32
==32199== NVPROF is profiling process 32199, command: ./vecadd 1048576 32
Finished! All 1048576 elements validate using 32 threads per block.
Took 5049 microseconds on the host
Took 4541 microseconds on the device, 1.11187x speedup
==32199== Profiling application: ./vecadd 1048576 32
==32199== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.85%  1.5758ms         1  1.5758ms  1.5758ms  1.5758ms  [CUDA memcpy DtoH]
 43.93%  1.5436ms         2  771.81us  767.70us  775.93us  [CUDA memcpy HtoD]
 10.26%  360.38us         1  360.38us  360.38us  360.38us  vector_add(int*, int*, int*, int)
  0.96%  33.856us         1  33.856us  33.856us  33.856us  [CUDA memset]
```

Review – CUDA Execution Model

nvcc – Compile CUDA applications, shares many flags with x86 compilers, use -arch to target specific GPU generations.

nvprof – Tool for analyzing CUDA applications, profile-driven optimization!

CUDA thread hierarchy: **thread**, **warp**, **thread block**, and **grid**

Access thread coordinates from running kernel via **`blockIdx.x`**, **`threadIdx.x`**, **`blockDim.x`**

Good to maximize parallelism of CUDA kernels over input, PCIe transfers can quickly become bottlenecks

CUDA Asynchronous Execution

CUDA Asynchronous Execution

Most APIs are synchronous by default.

- When you call `malloc`, you expect its return value to point to immediately addressable memory.

Commonly, high latency operations support some level of asynchrony:

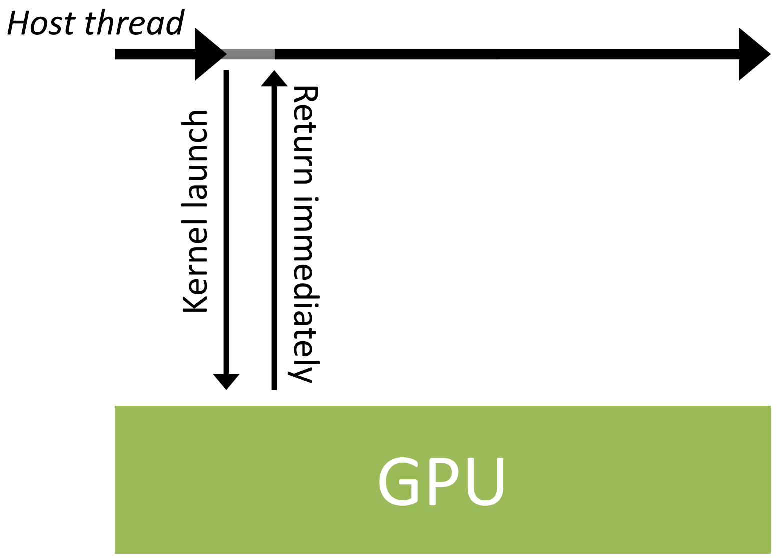
- POSIX AIO
- NodeJS/Javascript callbacks
- E-mail

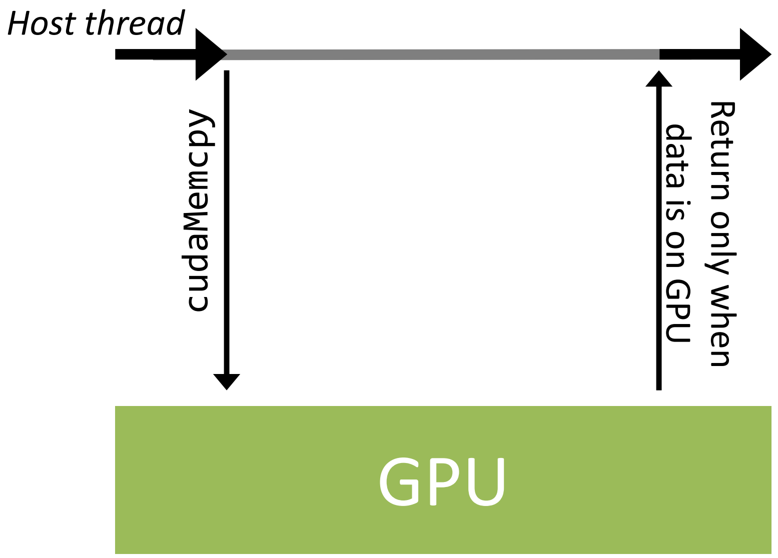
Overlapping useful work on-core with asynchronous work off-core can benefit performance.

CUDA Asynchronous Execution

CUDA supports both blocking and asynchronous APIs.

A kernel launch `<<<...>>>` is asynchronous. A `cudaMemcpy` is blocking.





CUDA Asynchronous Execution

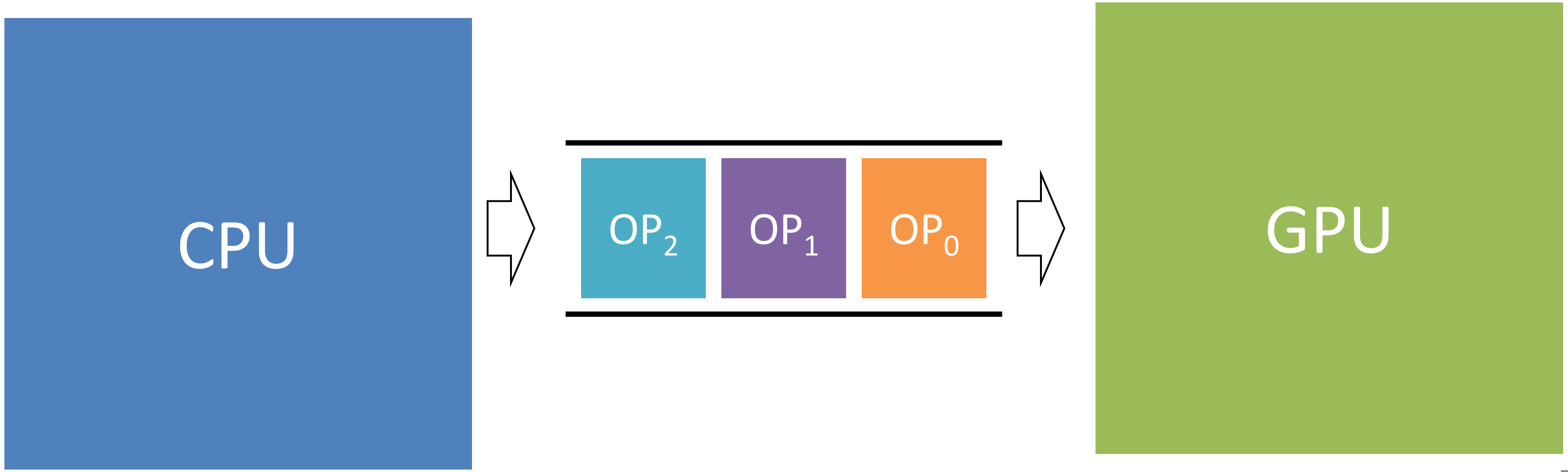
Some blocking APIs have asynchronous equivalents. Some don't.

Blocking API	Asynchronous API
`cudaMalloc`	
`cudaMemcpy`	`cudaMemcpyAsync`
	Kernel launch (`<<<...>>>`)
`cudaFree`	

CUDA Asynchronous Execution

The core of CUDA asynchrony is **CUDA streams**.

A CUDA stream defines a sequence of CUDA operations (e.g. kernel launch, memory copy, etc.) to be performed in order.



CUDA Asynchronous Execution

Streams must be explicitly created by the programmer:

```
cudaStream_t stream;
cudaError_t cudaStreamCreate(cudaStream_t *stream);
cudaError_t cudaStreamDestroy(cudaStream_t *stream);
```

CUDA Asynchronous Execution

Every asynchronous operation is associated with a CUDA stream.

If the CUDA stream is not explicitly set, a **default stream** is used.

Implicit default stream

```
kernel<<<...>>>(...);
cudaMemcpyAsync(...);
```

Explicit stream

```
kernel<<<..., stream>>>(...);
cudaMemcpyAsync(..., stream);
```

CUDA Asynchronous Execution

Stream status can be queried or blocked on.

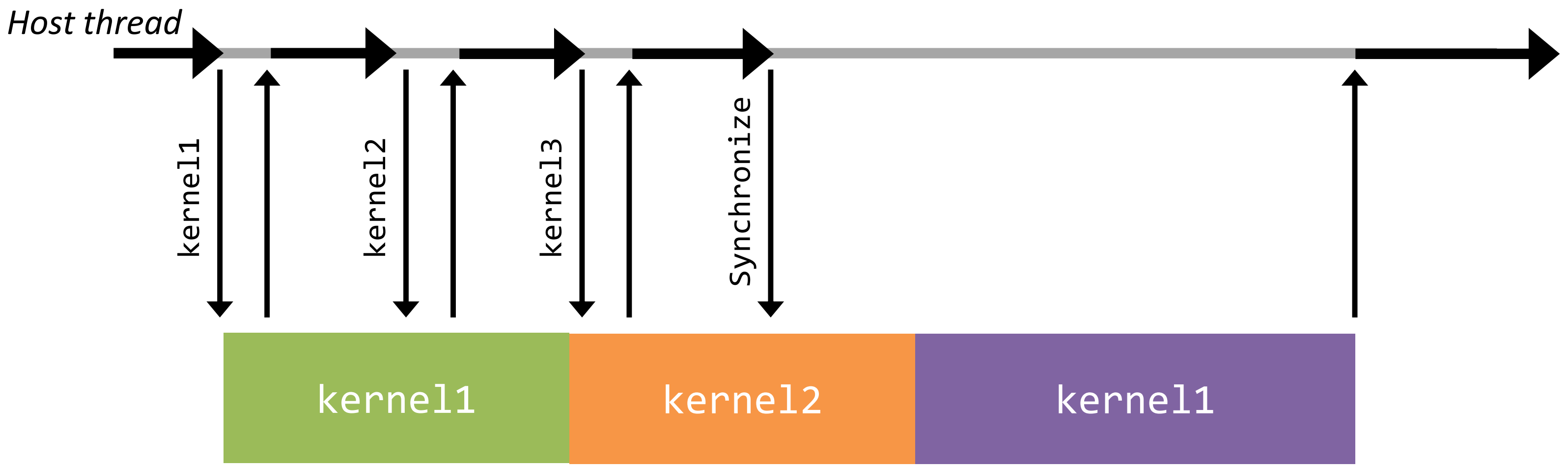
Implicitly includes all past operations in that stream.

```
cudaStream_t stream;
cudaStreamCreate(&stream);
kernel<<<..., stream>>>(...);
cudaStreamSynchronize(stream);
```

```
cudaError_t cudaStreamSynchronize(cudaStream_t stream);
cudaError_t cudaStreamQuery(cudaStream_t stream);
```

CUDA Asynchronous Execution

```
kernel1<<<..., stream>>>(...);
kernel2<<<..., stream>>>(...);
kernel3<<<..., stream>>>(...);
cudaStreamSynchronize(stream);
```

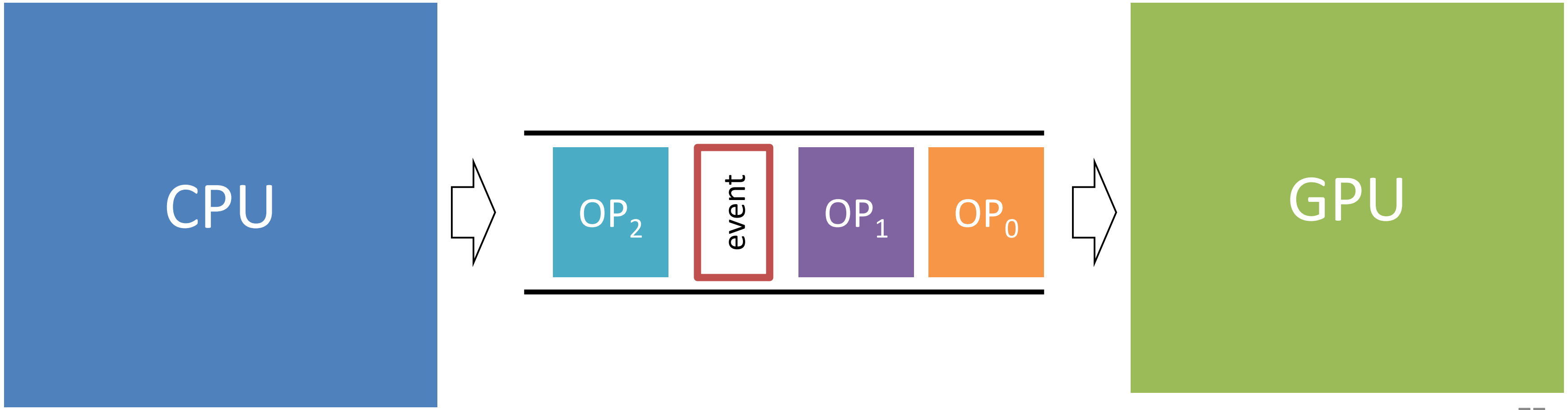


CUDA Asynchronous Execution

Also possible to query specific points in time within a stream using **CUDA Events**.

Events are inserted in to streams and satisfied as they exit the stream.

Enable querying of all operations inserted in to the stream prior to the event.



CUDA Asynchronous Execution

Like streams, events can be created, destroyed, queried, and blocked on.

```
cudaEvent_t event;
cudaError_t cudaEventCreate(cudaEvent_t *event,
        unsigned int flags);
cudaError_t cudaEventDestroy(cudaEvent_t event);
cudaError_t cudaEventQuery(cudaEvent_t event);
cudaError_t cudaEventSynchronize(cudaEvent_t event);
```

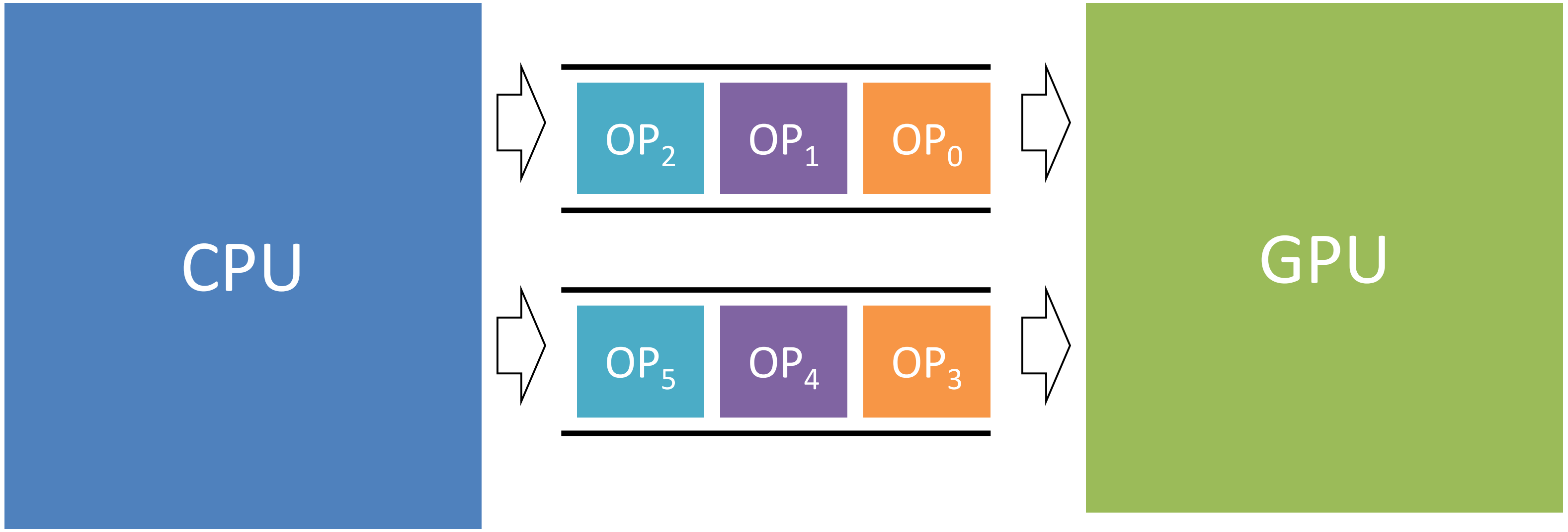
Placing an event in a stream:

```
cudaError_t cudaEventRecord(cudaEvent_t event,
        cudaStream_t stream);
```

CUDA Asynchronous Execution

Synchronizing on all operations issued to a given device is possible with `cudaDeviceSynchronize`.

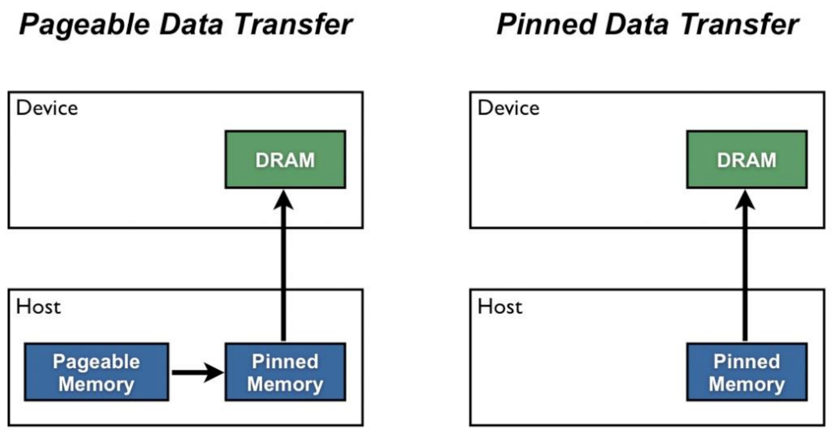
```
cudaError_t cudaDeviceSynchronize();
```



Special Note on cudaMemcpyAsync

Host allocations transferred through `cudaMemcpyAsync` must be page-locked/pinned.

CUDA provides `cudaMallocHost` for page-locked host allocations.

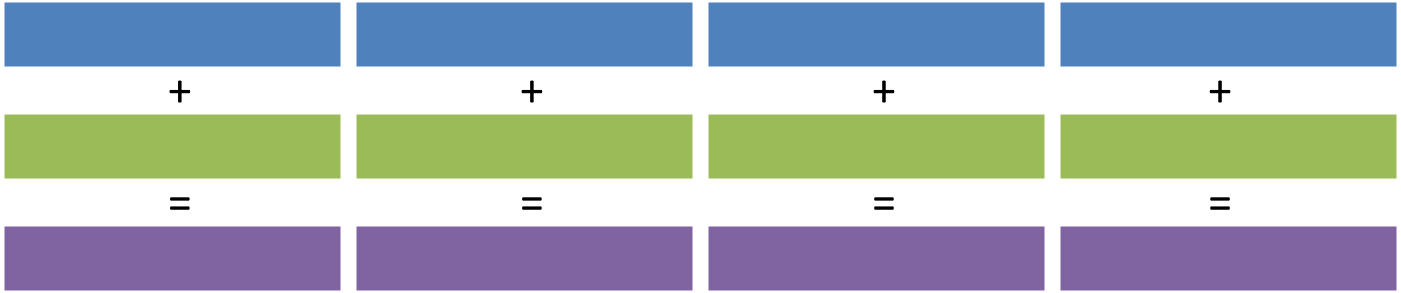


```
cudaError_t cudaMallocHost(void **ptr, size_t size);
```

https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

CUDA Asynchronous Execution

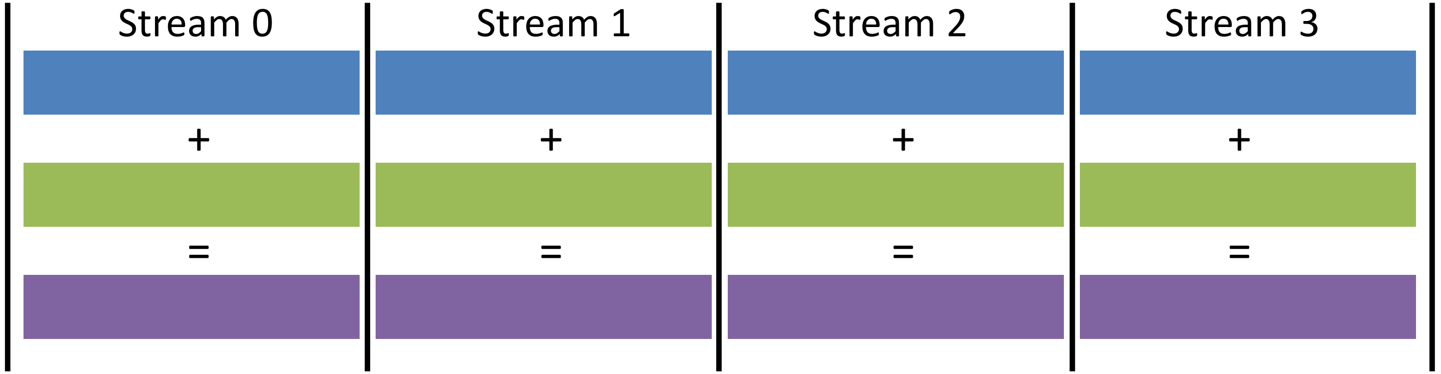
`$HOME/bootcamp-gpu/src/02_vecadd_streams/vecadd.cu` contains an example of chunked, synchronous vector addition.



```
$ ./vecadd <N> <nchunks>
```

Hands On – CUDA Streams

Complete the template in `$HOME/bootcamp-gpu/src/02_vecadd_streams/vecadd.cu` based on the TODOs contained to use a different stream on each vector chunk.



```
$ ./vecadd 4194304 4 # Vector of 4,194,304 elements, 4 chunks
```

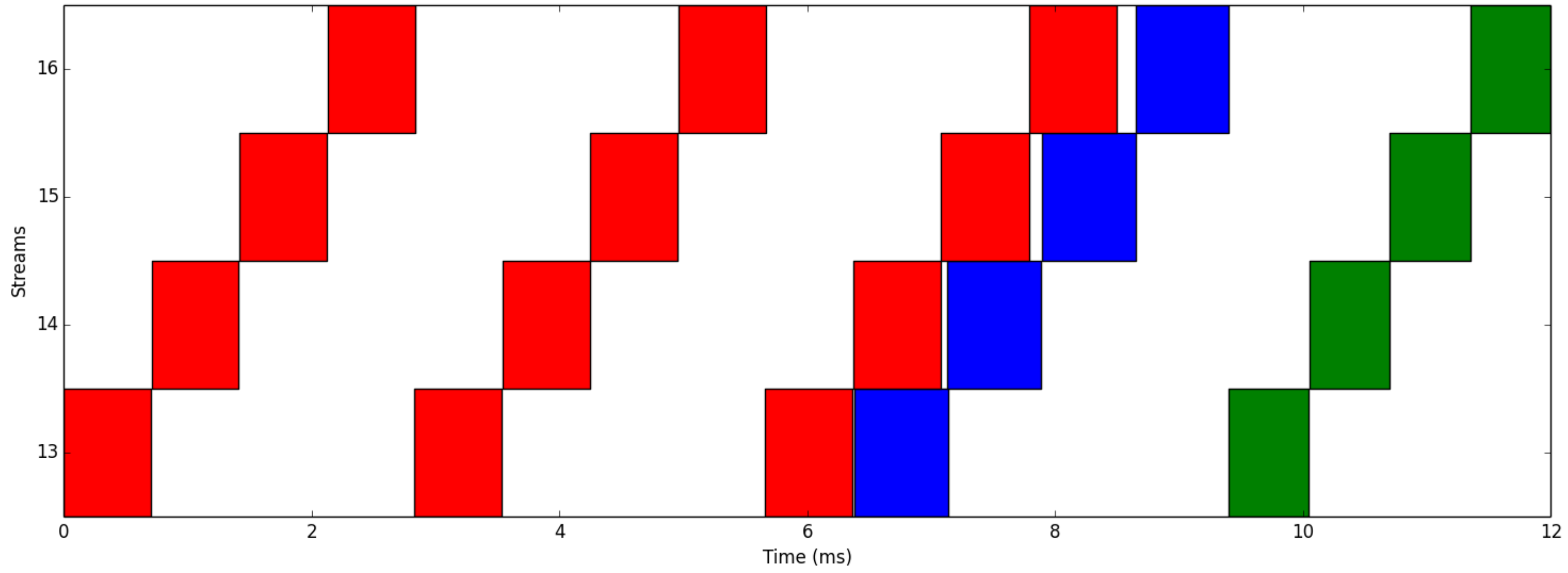
Do multiple streams significantly improve performance for this example?

Hands On – CUDA Streams

Try using nvprof’s GPU tracing mode to understand stream interleaving.

```
$ nvprof --print-gpu-trace ./vecadd_solution 4194304 1

$ nvprof --print-gpu-trace ./vecadd_solution 4194304 4
```

Hands On – CUDA Streams



🟥 Copy to device

🟦 Kernel

🟩 Copy to host

Review – CUDA Asynchrony

A **CUDA stream** defines a sequence of CUDA operations (e.g. kernel launch, memory copy, etc.) to be performed in order.

CUDA events are inserted in to streams and satisfied as they exit the stream, mark a specific point-in-time in the processing of the operations in that stream.

Using asynchronous APIs requires an understanding of streams (and events), can benefit performance by enabling host-device, host-copy, device-copy overlaps.

```
cudaStreamCreate, cudaStreamDestroy, cudaStreamSynchronize,
cudaStreamQuery, cudaEventCreate, cudaEventDestroy,
cudaEventQuery,  cudaEventSynchronize, cudaEventRecord,
cudaDeviceSynchronize, cudaMallocHost
```

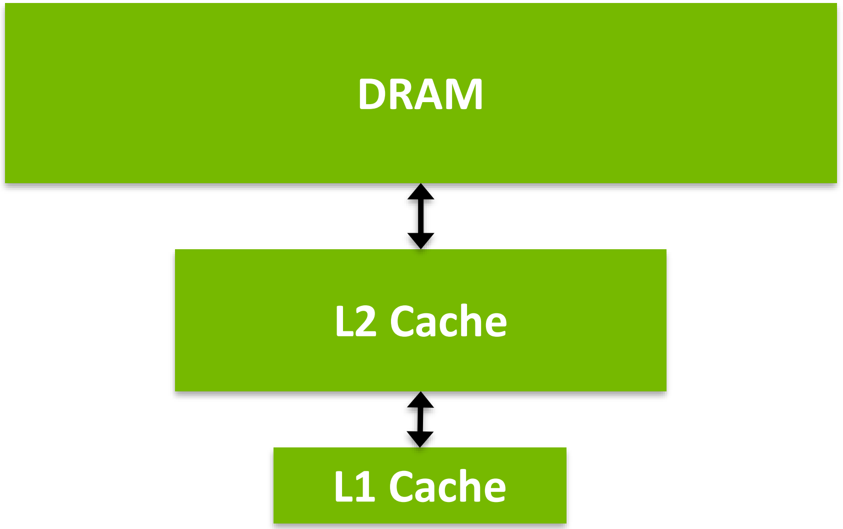
CUDA Memory Hierarchy

Memory Hierarchies

Up until now, we've glossed over efficient data access in CUDA kernels.

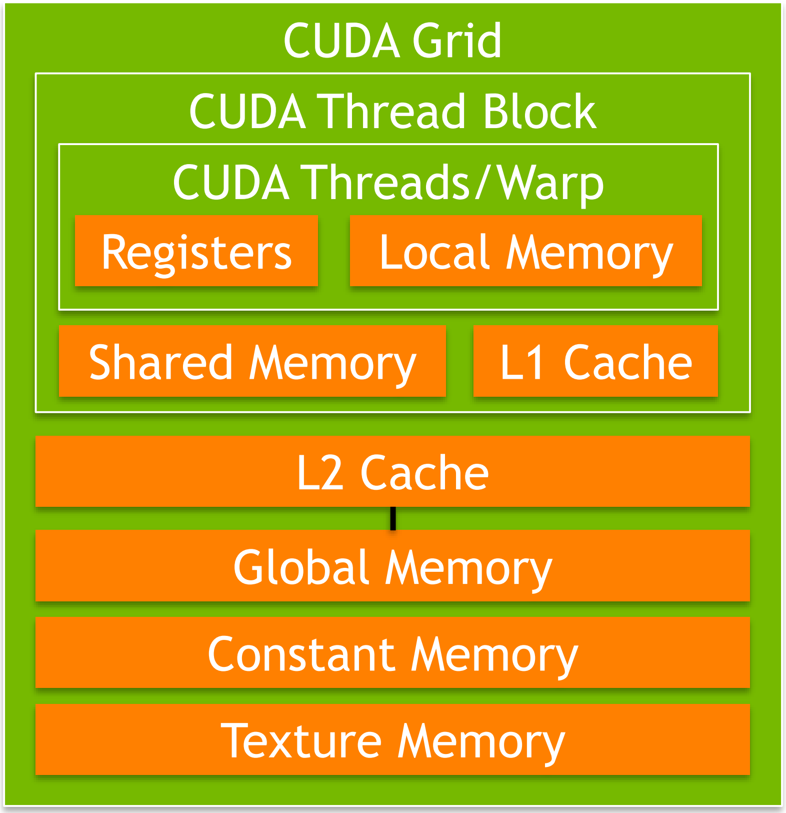
On CPU, this generally involves optimizing for cache line locality.

- A memory hierarchy emulates a large amount of low-latency memory



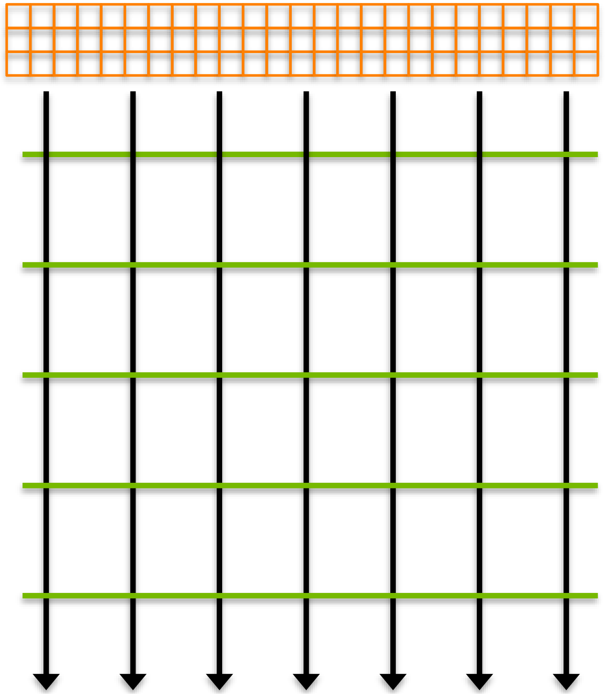
CUDA Memory Hierarchy

- The CUDA Memory Hierarchy is more complex than the CPU's
  - Many different types of memory, each with special-purpose characteristics
  - More explicit control over data movement



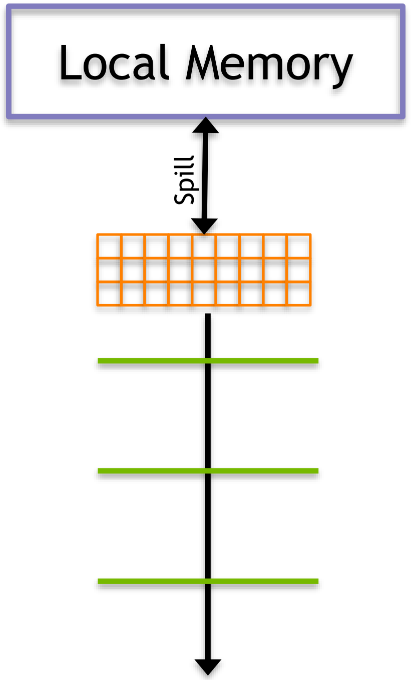
CUDA Memory Hierarchy

- **Registers**
  - Lowest latency memory space on the GPU
  - Private to each CUDA thread
  - Constant pool of registers per-SM divided among threads in resident thread blocks
  - Architecture-dependent limit on number of registers per thread
  - Registers are not explicitly used by the programmer, implicitly allocated by the compiler
  - `nvcc -maxrregcount` allows you to limit # registers per thread



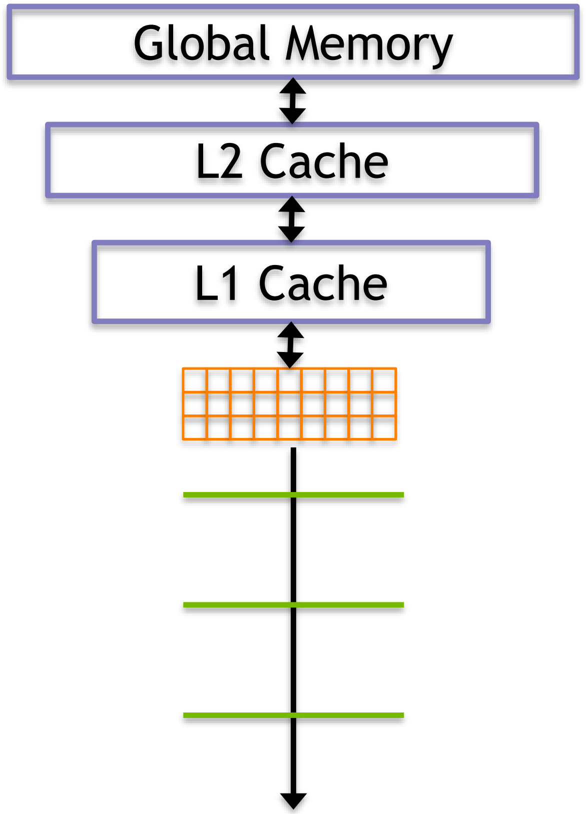
CUDA Memory Hierarchy

- **Local Memory**
  - When registers are exhausted, variables spill to local memory
  - Variables likely to be placed in local memory: large local structures or arrays, local arrays whose indices cannot be determined at compile-time
  - Local memory is not physical, variables stored in local memory are spilled to Global Memory, L1 cache, or L2 cache
  - Not explicitly controlled by programmer



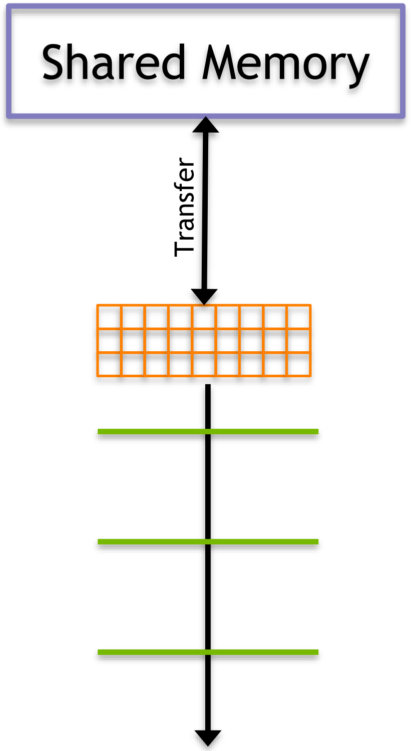
CUDA Memory Hierarchy

- **GPU Caches**
  - Behaviour of GPU caches is architecture-dependent
  - Per-SM L1 cache stored on-chip
  - Per-GPU L2 cache stored off-chip, caches values for all SMs
  - Due to parallelism of accesses, GPU caches can be difficult to reason about



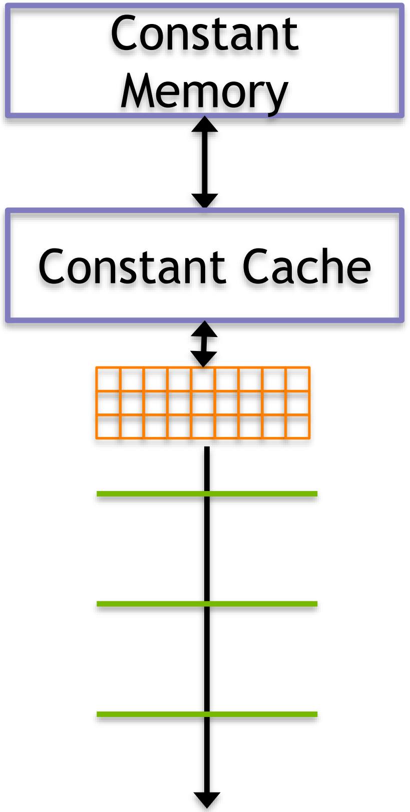
CUDA Memory Hierarchy

- **Shared Memory**
  - Declared with the `__shared__` keyword
  - Low-latency, high bandwidth
  - Shared by all threads in a thread block
  - Explicitly allocated and managed by the programmer, manual L1 cache
  - Stored on-SM, same physical memory as the GPU L1 cache
  - On-SM memory is statically partitioned between L1 cache and shared memory



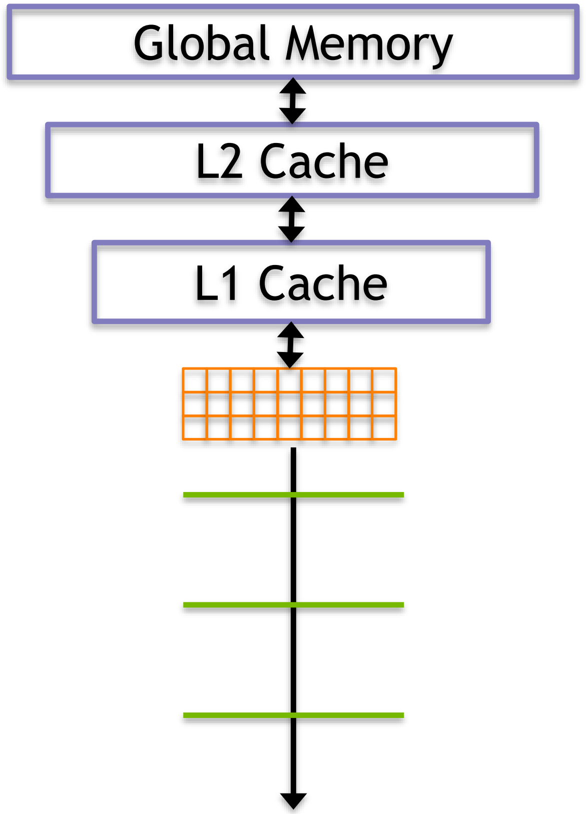
CUDA Memory Hierarchy

- **Constant Memory**
  - Declared with the `__constant__` keyword
  - Read-only
  - Limited in size: 64KB
  - Stored in device memory (same physical location as Global Memory)
  - Cached in a per-SM constant cache
  - Optimized for all threads in a warp accessing the same memory cell



CUDA Memory Hierarchy

- **Global Memory**
  - Large, high-latency memory
  - Stored in device memory (along with constant and texture memory)
  - Can be declared statically with `__device__`
  - Can be allocated dynamically with `cudaMalloc`
  - Explicitly managed by the programmer
  - Optimized for all threads in a warp accessing neighbouring memory cells



Review - CUDA Memory Spaces

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

CUDA Kernel Synchronization

Kernel Synchronization

Synchronization in CUDA is closely tied with the CUDA memory hierarchy.

Relative to CPU programming models, synchronization options are constrained in CUDA.

- CUDA is designed to be highly scalable and portable across generations
- Lots of synchronization negates both of these principles

There is no global synchronization in CUDA (i.e. no `#pragma omp barrier`) – global synchronization requires kernel termination.

Instead, CUDA offers local synchronization and some global memory fences.

- For performance reasons, use of these should still be minimized

Kernel Synchronization

Function	Action	Scope	Notes
`__syncthreads()`	Barrier	Thread block	Most commonly useful function.
`__threadfence_block()`	Memory fence	Thread block	Applies to shared and global mem.
`__threadfence()`	Memory fence	Device	Same as `__threadfence_block()`, but also a global memory write fence relative to all threads on the device.
`__threadfence_system()`	Memory fence	Whole system	Same as `__threadfence_block()`, but also a global memory write fence relative to all threads on the device and host and read fence for calling thread.

Review - Kernel Synchronization

Good chance you'll only ever use `__syncthreads()`

Avoid synchronization at all costs.

Global synchronization by kernel termination.

Optimizing CUDA Memory Accesses

CPUs rely heavily on hardware-managed caches for performance.

GPU caching is a more challenging problem

- Thousands of threads cooperating on a problem
- Difficult to predict the next round of accesses for all threads

For efficient global memory access, GPUs instead rely on:

1. Large device memory bandwidth
2. Aligned and coalesced memory access patterns
3. Maintaining sufficient pending I/O operations to keep the memory bus saturated and hide global memory latency

Optimizing CUDA Memory Accesses

Aligned and coalesced global memory accesses are key to optimizing an application’s use of global memory bandwidth.

1. **Coalesced**: threads in a warp reference memory addresses that can all be serviced by a single global memory transaction (think of a memory transaction as the process of bring a cache line into the cache)
2. **Aligned**: the global memory accesses by threads within a warp start at an address boundary that is an even multiple of the size of a global memory transaction

Optimizing CUDA Memory Accesses

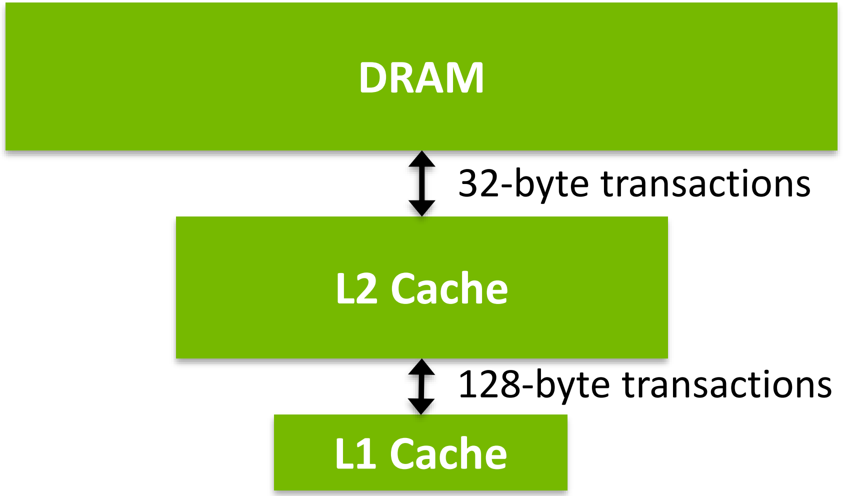
Global memory transactions are either 32 or 128 contiguous bytes.

Up to Maxwell, size of memory transaction depends on caches it passes through.

- **L1 + L2** - 128 bytes, **L2 only** - 32 bytes

With Maxwell/Pascal, all transactions are 32 bytes.

Information in the "Volta Tuning Guide" is sparse at this time, but expect similar behaviour to Pascal.



Optimizing CUDA Memory Accesses

Which caches a global memory transaction passes through depends on GPU architecture and the type of access (read vs. write).

Architecture	Reads Cacheable in L1?	Cached by Default?
Fermi	Yes	Yes
Kepler	No	No
Kepler K40 + Maxwell	Yes	No
Pascal P100	Yes	Yes
Pascal P104	Yes	No
Volta	Yes	Yes

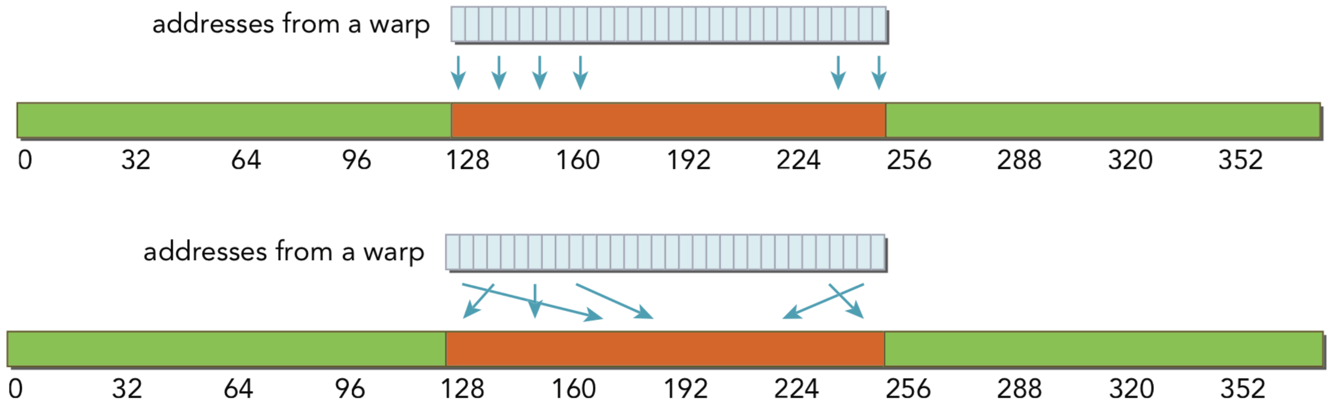
Global memory reads cached in L2 if not in L1

Architecture	Writes Cacheable in L1?	Cached by Default?
Fermi	No	No
Kepler	No	No
Kepler K40 through Volta	No	No
Volta	Yes	Yes

Global memory writes cached in L2, up until Volta.

Optimizing CUDA Memory Accesses

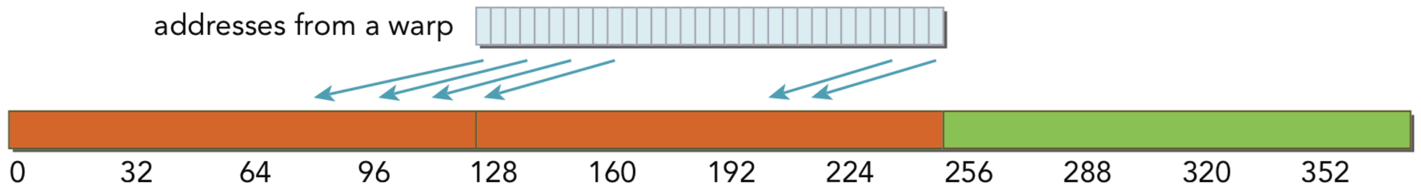
Aligned and Coalesced Memory Access (*w/ L1 cache*)



With 128-byte transactions, a single transaction is required and all of the loaded bytes are used

Optimizing CUDA Memory Accesses

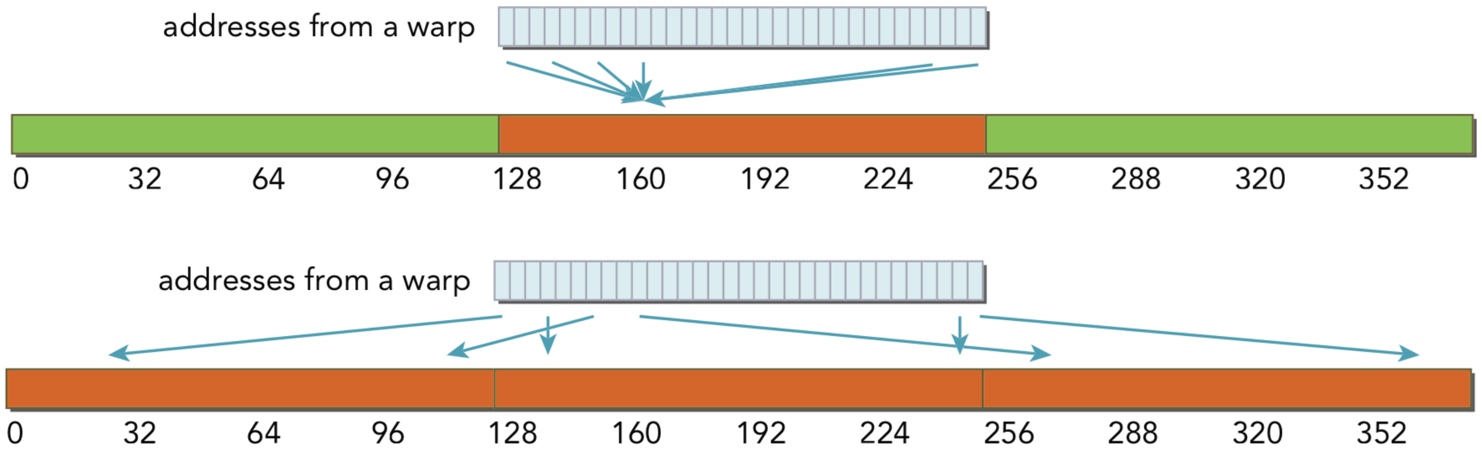
Misaligned and Coalesced Memory Access (*w/ L1 cache*)



With 128-byte transactions, two memory transactions are required to load all requested bytes. Only half of the loaded bytes are used.

Optimizing CUDA Memory Accesses

Misaligned and Uncoalesced Memory Access (*w/ L1 cache*)



With uncoalesced loads, many more bytes loaded than requested.

Optimizing CUDA Memory Accesses

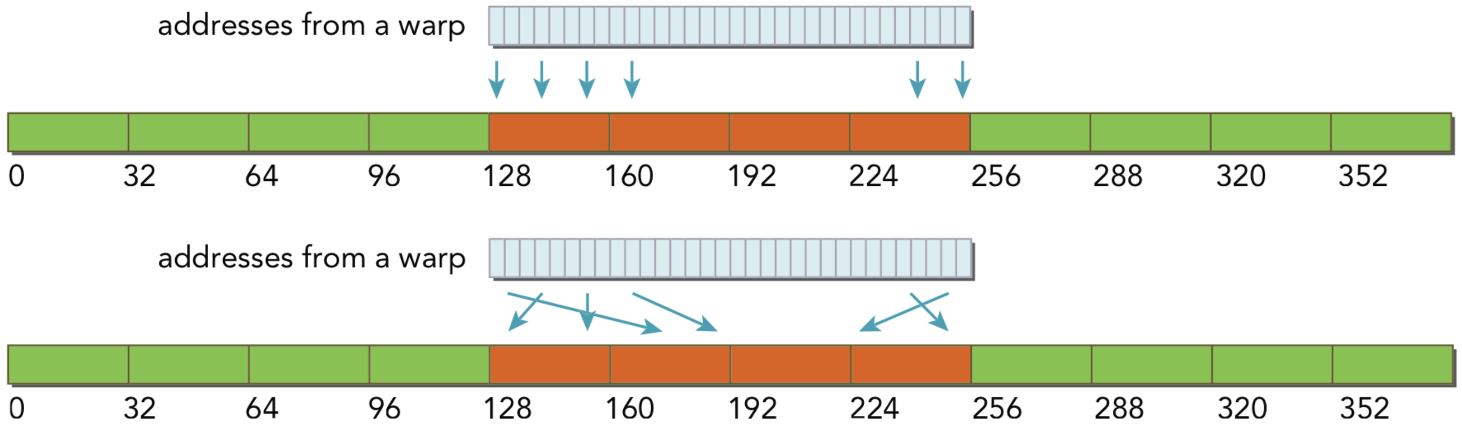
Memory accesses that are not cached in L1 are serviced by 32-byte transactions

This can improve memory bandwidth utilization

However, the L2 cache is device-wide, higher latency than L1, and still relatively small ➔ many applications may take a performance hit if L1 cache is not used for reads

Optimizing CUDA Memory Accesses

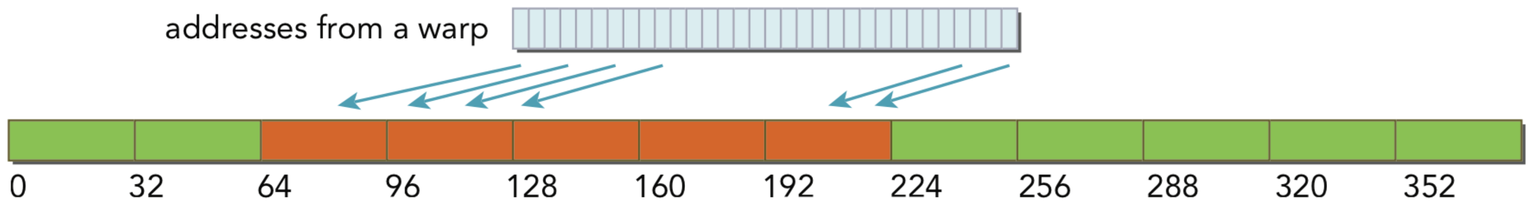
Aligned and Coalesced Memory Access (*w/o L1 cache*)



With 32-byte transactions, four transactions are required and all of the loaded bytes are used

Optimizing CUDA Memory Accesses

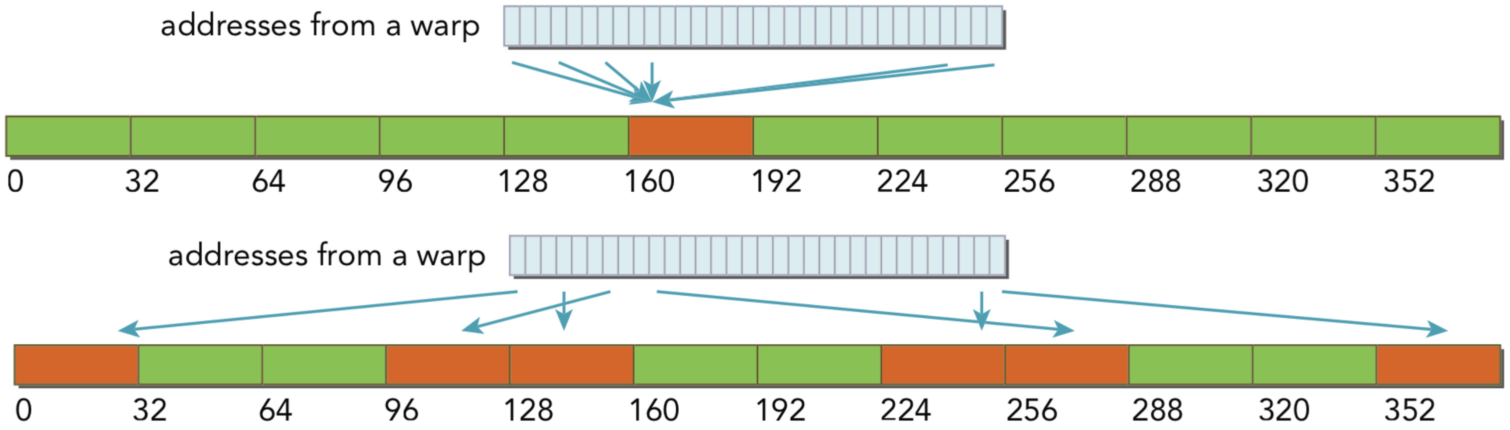
Misaligned and Coalesced Memory Access (*w/o L1 cache*)



With 32-byte transactions, extra memory transactions are still required to load all requested bytes but the number of wasted bytes is likely reduced compared to 128-byte transactions.

Optimizing CUDA Memory Accesses

Misaligned and Uncoalesced Memory Access (*w/o L1 cache*)

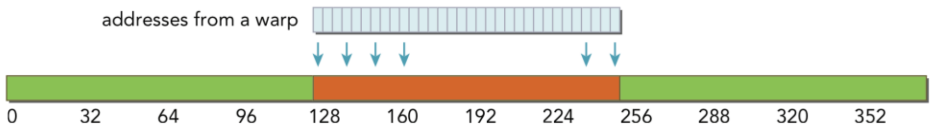


With uncoalesced loads, more bytes loaded than requested but better efficiency than with 128-byte transactions.

Hands On – Memory Access Optimization

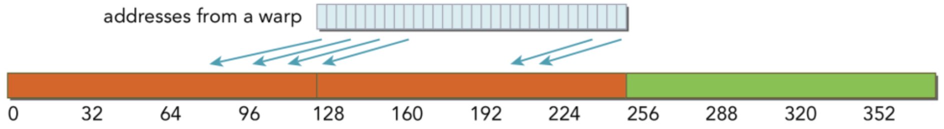
To illustrate these points, we'll experiment with a variation on the vector add micro-benchmark under `$HOME/bootcamp-gpu/src/03_vecadd_misaligned`.

`vector_add`

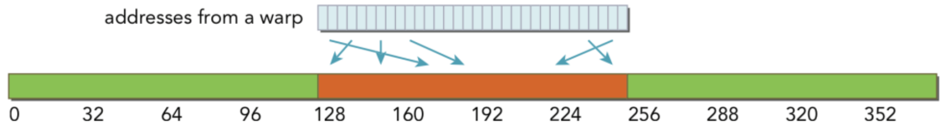


`vector_add_read_offset`

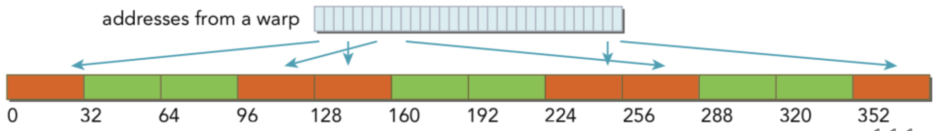
`vector_add_write_offset`



`vector_add_weirdly_coalesced`



`vector_add_not_coalesced`



Hands On – Memory Access Optimization

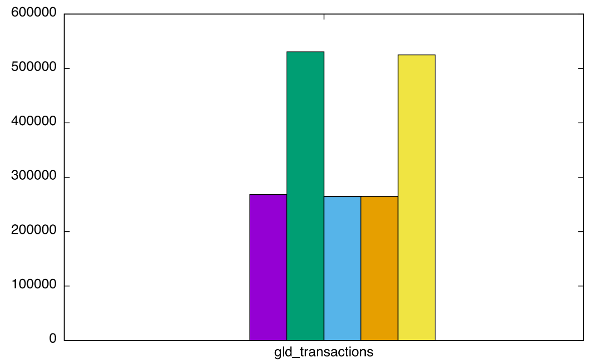
Experiment with several new metrics from nvprof:

1. gld_transactions: # of global memory load transactions
2. gst_transactions: # of global memory store transactions
3. gld_transactions_per_request: mean # of gld txs issued to satisfy a warp's request
4. gst_transactions_per_request: mean # of gst txs issued to satisfy a warp's request
5. gld_efficiency: gld_requests / gld_transactions
6. gst_efficiency: gst_requests / gst_transactions

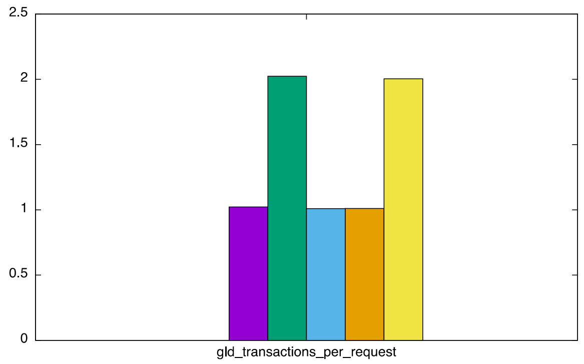
```
$ nvprof --metrics gld_transactions ./vecadd 4194304
$ nvprof --metrics gst_transactions ./vecadd 4194304
$ ...
```

What do you observe?

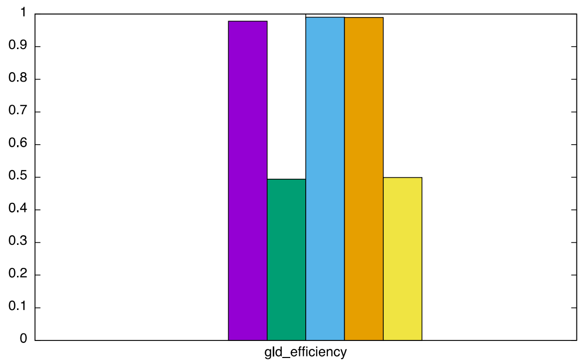
Hands On – Memory Access Optimization



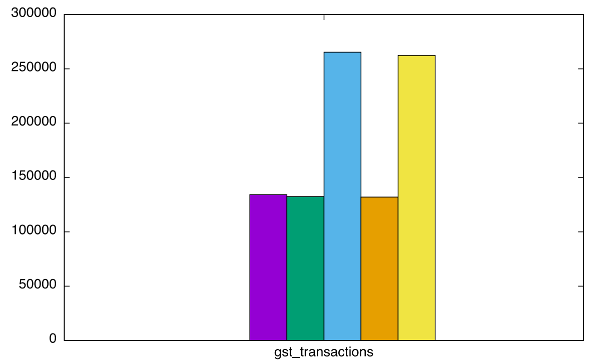
Misaligned, uncoalesced reads ↑ gld_transactions



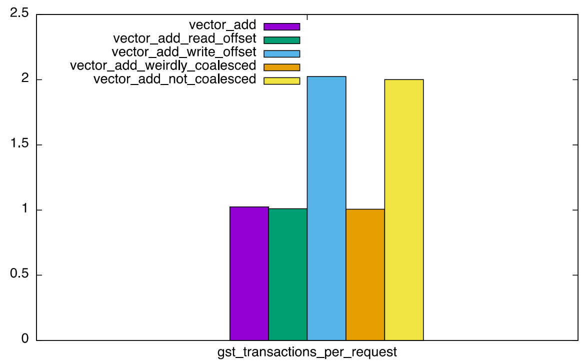
Misaligned, uncoalesced reads
↑ gld_transactions_per_request



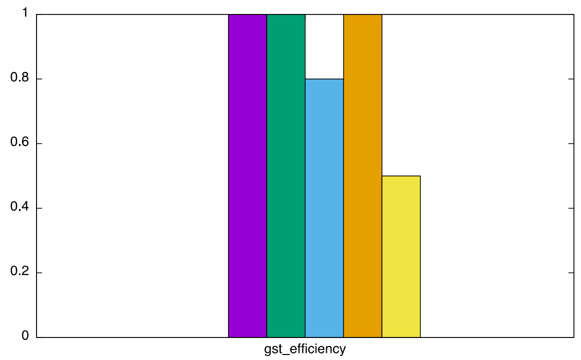
Misaligned, uncoalesced reads ↓ gld_efficiency



Misaligned, uncoalesced writes ↑ gst_transactions



Misaligned, uncoalesced writes
↑ gst_transactions_per_request



Misaligned, uncoalesced writes ↓ gst_efficiency

Review - Optimizing CUDA Memory Access

Seek **aligned** and **coalesced** global memory accesses when optimizing CUDA kernel performance.

The way in which a load/store is serviced depends on the cache(s) it passes through.

Aligned and coalesced accesses reduce the number of transactions necessary and the efficiency of each of those transactions (i.e. # bytes requested / # bytes loaded).

CUDA Atomic Operations (Briefly)

Atomic Operations

Atomic operations are a special class of mathematical operations in computing

- An atomic operation is performed uninterruptedly, so that there is no interference from other threads
- When a thread's atomic operation has completed, it can be certain its requested changes have been made without interference from other threads
- Atomic operations are particularly useful on the massively parallel GPU

Atomic operations may improve correctness, but can have a detrimental impact on performance

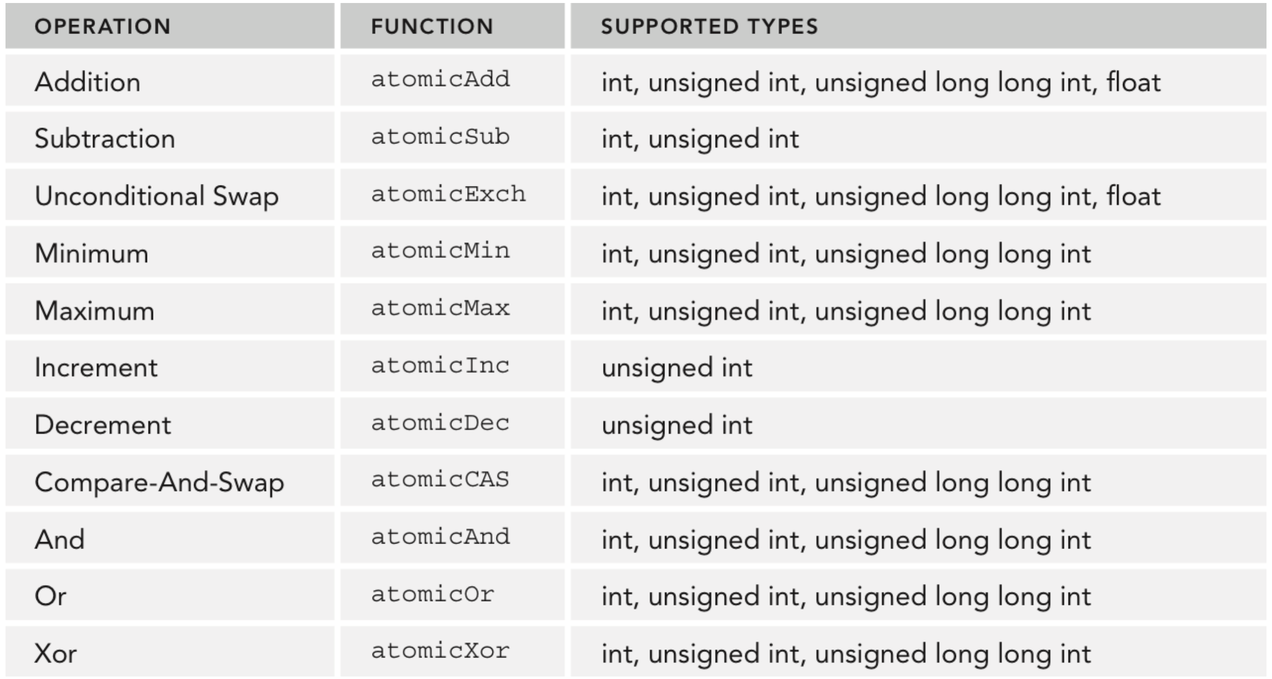
- Out of thousands of threads, only one can succeed in accessing a shared variable
- Atomicity requires reading and writing DRAM, no caching allowed

Atomic Operations

```
__global__ void sumAll(int *in_array, int N, int *out_scalar) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        // out_scalar += out_array[i] would cause massive data race
        atomicAdd(out_scalar, in_array[i]);
    }
}
```

Atomic Operations Summary

CUDA supports a variety of atomic operations. CAS can be used to build your own.



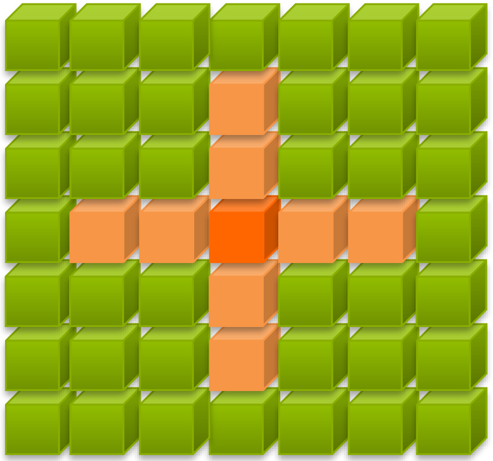
OPERATION	FUNCTION	SUPPORTED TYPES
Addition	`atomicAdd`	int, unsigned int, unsigned long long int, float
Subtraction	`atomicSub`	int, unsigned int
Unconditional Swap	`atomicExch`	int, unsigned int, unsigned long long int, float
Minimum	`atomicMin`	int, unsigned int, unsigned long long int
Maximum	`atomicMax`	int, unsigned int, unsigned long long int
Increment	`atomicInc`	unsigned int
Decrement	`atomicDec`	unsigned int
Compare-And-Swap	`atomicCAS`	int, unsigned int, unsigned long long int
And	`atomicAnd`	int, unsigned int, unsigned long long int
Or	`atomicOr`	int, unsigned int, unsigned long long int
Xor	`atomicXor`	int, unsigned int, unsigned long long int

Hands on with a Real World App

A (Slightly) More Complex Application

2D wavefront propagation (stencil)

```
for (int y = 0; y < ny; y++) {
  for (int x = 0; x < nx; x++) {

    for (int d = 1; d <= radius; d++) {
      div += c_coeff[d] * (curr[y_pos_offset] +
         curr[y_neg_offset] + curr[x_pos_offset] +
         curr[x_neg_offset]);
    }

    next[this_offset] = temp + div * vsq[this_offset];
  }
}
```



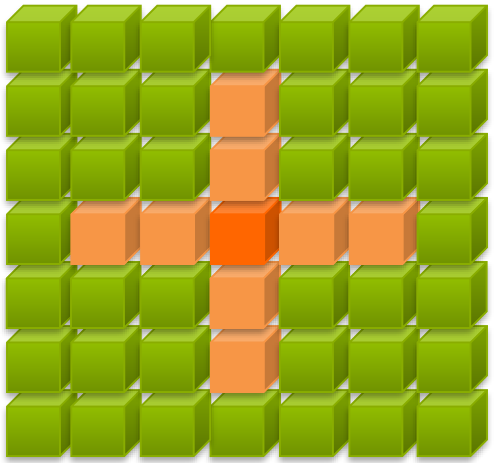
We'll use `iso2d` as a more realistic example to explore optimizations.

A More Complex Application

Provided reference code:

- `04_iso2d_seq/`: sequential C implementation

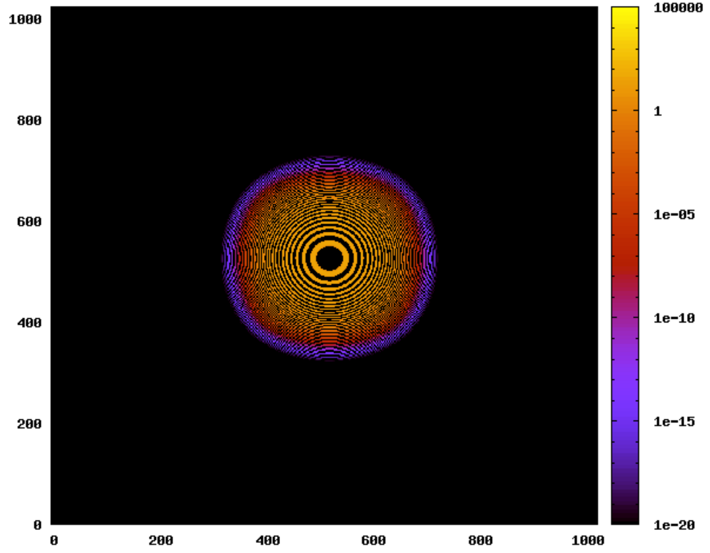
```
$ cd $HOME/bootcamp-gpu/src/04_iso2d_seq
$ make

$ ./iso2d_seq -x 1024 -y 1024 -i 800 -t
```



A More Complex Example

iso2d's –t flag will produce an output file to visualize for correctness.

```
$ ../iso2d_common/iso.sh –x 1024 –y 1024 –i snap.text
$ eog iso.png # Must use ssh –X to connect to DAVINCI
```



Hands On – A Quick Review of OpenMP

Let's start by examining the parallelism possible in `iso2d` using OpenMP.

Where could you add a `#pragma omp parallel for` in `iso2d.cpp` for the most parallelism?

Compare the performance of your OpenMP implementation to the provided sequential code by modifying `iso2d.cpp` in `04_iso2d_seq`.

Visualize the results to verify the correctness of your OpenMP implementation.

```
$ ./iso2d_omp -x 1024 -y 1024 -i 800
```

(If you're having trouble you can cheat and look in `05_iso2d_omp/`)

Hands On – OpenMP to CUDA

Next, your task is to port the OpenMP version of `iso2d` to CUDA, and check its performance.

`06_iso2d_cuda/iso2d.cu` contains TODOs to help in the port.

Use nvprof's Summary Mode to check where your implementation is spending time.

(If you're having trouble you can cheat and look in `iso2d_2dsolution.cu` or `iso2d_1dsolution.cu` for solutions using 2D and 1D thread blocks)

Hands On – OpenMP to CUDA

Use nvprof's Summary Mode to check where your implementation is spending time.

```
iso_r4_2x:   0.4557509422 s total, 0.0005638975 s/step,  1859.52 Mcells/s/step
==19313== Profiling application: ./iso2d_cuda_solution -x 1024 -y 1024 -i 800
==19313== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 98.94%  432.51ms       800  540.64us  537.07us  546.56us  fwd_kernel
  0.85%  3.7121ms       804  4.6160us  1.2150us  955.81us  [CUDA memcpy HtoD]
  0.21%  911.11us         1  911.11us  911.11us  911.11us  [CUDA memcpy DtoH]

==19313== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 82.61%  444.60ms       805  552.29us  9.3120us  1.2511ms  cudaMemcpy
 15.56%  83.736ms         4  20.934ms  144.61us  83.285ms  cudaMalloc
  1.18%  6.3430ms       800  7.9280us  7.1660us  28.247us  cudaLaunch
  0.29%  1.5825ms      6400     247ns     207ns  8.7500us  cudaSetupArgument
```

Hands On – OpenMP to CUDA

Dig deeper using nvprof metrics. We'll start by just looking at the ones we've tried in past examples.

`$ nvprof --metrics <metric> –x 1024 –y 1024 –i 800 –t` with:

1. ipc
2. sm_efficiency
3. alu_fu_utilization
4. gld_transactions
5. gst_transactions
6. gld_transactions_per_request
7. gst_transactions_per_request
8. gld_efficiency
9. gst_efficiency

Hands On – OpenMP to CUDA

Dig deeper using nvprof metrics. We'll start by just looking at the ones we've tried in past examples.

```
Invocations                             Metric Name                        Metric Description         Avg
Device "Tesla M2050 (0)"
        Kernel: fwd_kernel
        800                           sm_efficiency                   Multiprocessor Activity      99.92%
        800                                     ipc                              Executed IPC    1.594381
        800                        gld_transactions                  Global Load Transactions    19710600
        800                        gst_transactions                 Global Store Transactions     1051120
        800            gld_transactions_per_request      Global Load Transactions Per Request    1.978683
        800            gst_transactions_per_request     Global Store Transactions Per Request    2.004852
        800                          gld_efficiency             Global Memory Load Efficiency      50.71%
        800                          gst_efficiency            Global Memory Store Efficiency      80.00%
        800                     alu_fu_utilization      Arithmetic Function Unit Utilization    High (7)
```

iso2d

What does this kernel load?

Accessed	Coalesced	Aligned?	Frequency of Access
curr[this_offset]	Yes	Maybe	A few threads
next[this_offset]	Yes	Maybe	A few threads
curr[y_pos_offset]	Yes	Maybe	A few threads
curr[y_neg_offset]	Yes	Maybe	A few threads
curr[x_pos_offset]	Yes	Maybe	A few threads
curr[x_neg_offset]	Yes	Maybe	A few threads
vsq[this_offset]	Yes	Maybe	A few threads
c_coeff[d]	No	Usually Not	All threads

iso2d

Constant memory is optimized for broadcast access (whole warp reads same location).

Variables in CUDA constant memory must be declared statically:

```
__constant__ TYPE const_c_coeff[NUM_COEFF];
```

Can be referenced as normal arrays from CUDA kernels:

```
const_c_coeff[d]
```

Initialized using a special-purpose `cudaMemcpy` API:

```
cudaMemcpyToSymbol(const_c_coeff, c_coeff, NUM_COEFF * sizeof(TYPE));
```

Hands On – Constant Memory

Try your hand at optimizing the `iso2d` CUDA version using constant memory.

Re-analyze the overall performance and the same profiler metrics.

A template with TODOs (along with an example solution) is available in `07_iso2d_cuda_cmem`.

Hands On – Constant Memory

Try your hand at optimizing the `iso2d` CUDA version using constant memory.

Re-analyze the overall performance and the same profiler metrics.

Version	Performance (Mcells/s/step)	Improvement
Sequential	73.35	
OpenMP	275.20	3.75
CUDA	2,708.77	9.84
+ cmem	2,812.53	1.04

```
-x 4096 -y 4096 -i 1000
```

Hands On – Constant Memory

Try your hand at optimizing the `iso2d` CUDA version using constant memory.

Re-analyze the overall performance and the same profiler metrics.

```
Invocations                        Metric Name                        Metric Description         Avg

Device "Tesla M2050 (0)"
        Kernel: fwd_kernel
        800                      sm_efficiency                   Multiprocessor Activity      99.90%
        800                                ipc                              Executed IPC    1.594887
        800                   gld_transactions                  Global Load Transactions    19714799
        800                   gst_transactions                 Global Store Transactions     1060304
        800       gld_transactions_per_request      Global Load Transactions Per Request    1.976322
        800       gst_transactions_per_request     Global Store Transactions Per Request    2.016384
        800                     gld_efficiency            Global Memory Load Efficiency      50.60%
        800                     gst_efficiency           Global Memory Store Efficiency      80.00%
        800                alu_fu_utilization   Arithmetic Function Unit Utilization      High (7)
```

No significant change in metrics

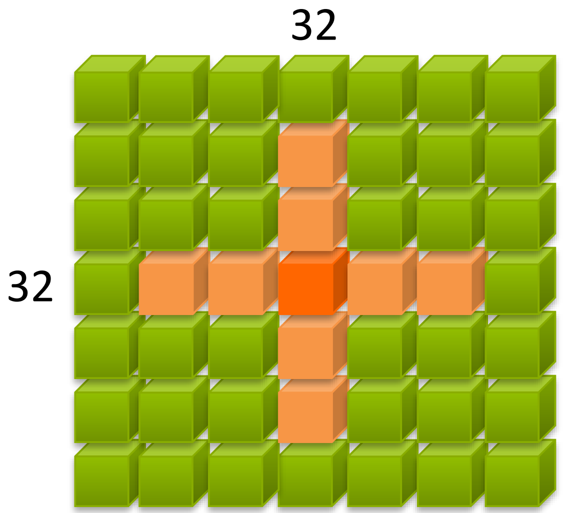
Hands On – Constant Memory

Back to the drawing board.

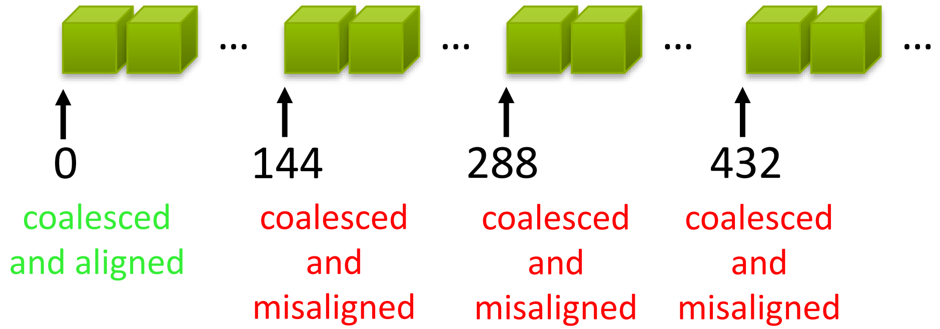
Accessed	Coalesced	Aligned?	Frequency of Access
curr[this_offset]	Yes	Maybe	A few threads
next[this_offset]	Yes	Maybe	A few threads
curr[y_pos_offset]	Yes	Maybe	A few threads
curr[y_neg_offset]	Yes	Maybe	A few threads
curr[x_pos_offset]	Yes	Maybe	A few threads
curr[x_neg_offset]	Yes	Maybe	A few threads
vsq[this_offset]	Yes	Maybe	A few threads
c_coeff[d]	No	Usually Not	All threads

Hands On – Access Alignment

Depending on the width of the 2D grid we're applying our stencil to, there may be many accesses that are mis-aligned.

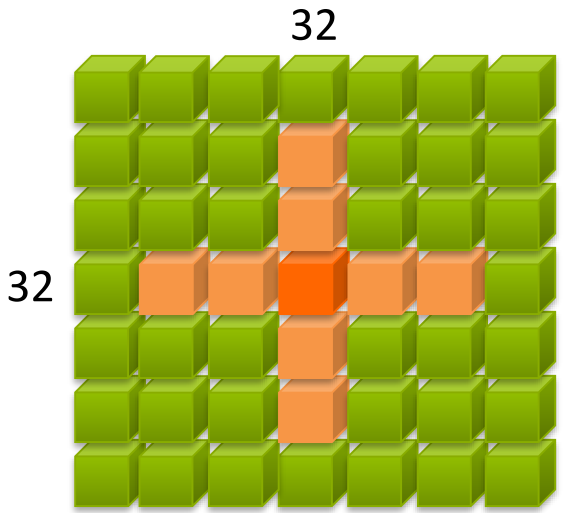


(2 + 32 + 2) * sizeof(float) = 144 bytes per row



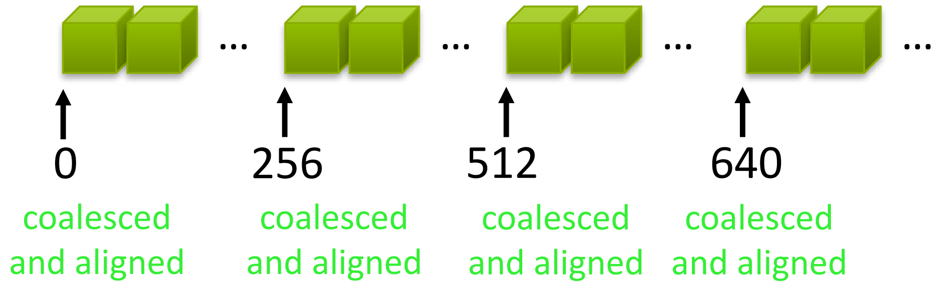
Hands On – Access Alignment

Depending on the width of the 2D grid we're applying our stencil to, there may be many accesses that are mis-aligned.



If we pad each row out to an even multiple of 128 bytes…

(2 + 32 + 2) * sizeof(float) + **112** = 256 bytes per row



Hands On – Access Alignment

Depending on the width of the 2D grid we're applying our stencil to, there may be many accesses that are mis-aligned.

By padding rows in the grid to be a length evenly divisible by 128 bytes, we can improve the # of aligned accesses (thereby reducing transactions).

Try writing a version of your constant memory version of `iso2d` that pads all rows of `curr`, `next`, and `vsq` out to 128 bytes.

Re-analyze the overall performance and the same profiler metrics.

`08_iso2d_cuda_aligned` contains a starting template and example solution.

Hands On – Access Alignment

Version	Performance (Mcells/s/step)	Improvement
Sequential	73.35	
OpenMP	275.20	3.75
CUDA	2,708.77	9.84
+ cmem	2,812.53	1.04
+ aligned	2944.26	1.05

```
-x 4096 -y 4096 -i 1000
```

Hands On – Access Alignment

Metric	Old Value	New Value
sm_efficiency	99.92%	99.90%
ipc	1.594381	1.717437
gld_transactions	19710600	14179671
gst_transactions	1051120	525216
gld_transactions_per_request	1.978683	1.423451
gst_transactions_per_request	2.004852	1.001770
gld_efficiency	50.71%	70.25%
gst_efficiency	80.00%	100.00%
alu_fu_utilization	High (7)	High (7)

Hands On – nvprof Throughput Metrics

Let’s collect some new metrics:

1. gld_throughput: Achieved throughput for global memory accesses
2. dram_read_throughput: Throughput between DRAM and L2
3. l2_read_throughput: Throughput through L2 cache

Hands On – nvprof Throughput Metrics

1. gld_throughput: Achieved throughput for global memory accesses
2. dram_read_throughput: Throughput between DRAM and L2
3. l2_read_throughput: Throughput through L2 cache

Massive discrepancy?

```
Kernel: fwd_kernel(float*, float*, float*, int, int, int, int)
  gld_throughput                 Global Load Throughput          312.02GB/s
  dram_read_throughput           Device Memory Read Throughput   53.643GB/s
  l2_read_throughput             L2 Throughput (Reads)           125.45GB/s
```

iso2d

Throughput metrics suggest that we're hitting a lot in L1 cache.

Can confirm using the `l1_cache_global_hit_rate` metric.

```
Metric Name                        Metric Description          Avg
Device "Tesla M2050 (0)"
        Kernel: fwd_kernel
l1_cache_global_hit_rate           L1 Global Hit Rate       64.03%
```

Good: we're getting lots of locality benefits.
Bad: L1 cache may be a bottleneck, losing some of our on-chip memory to shared memory.

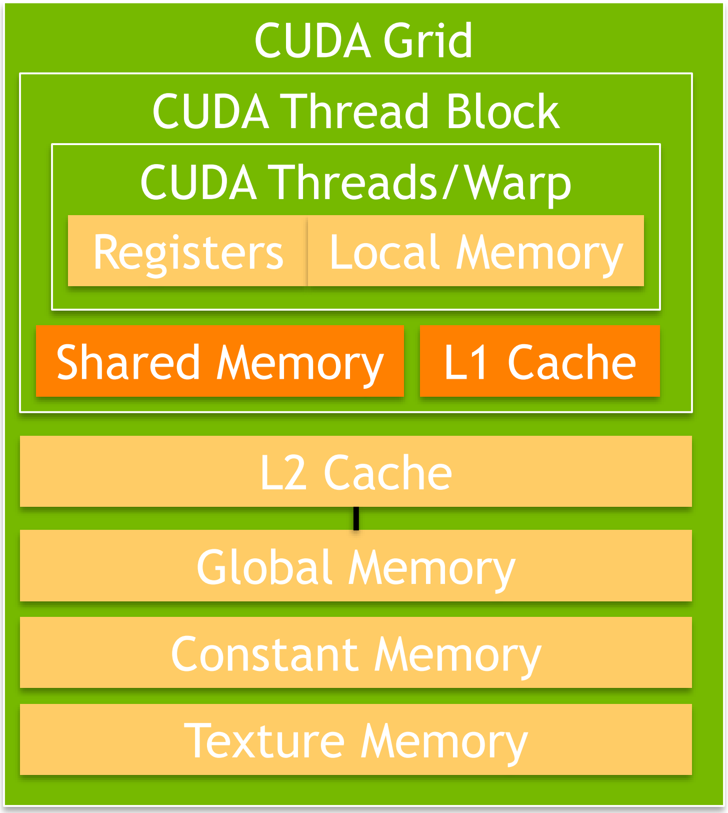
iso2d

Original discussion of shared memory:

- On-chip, low latency, high bandwidth
- Same physical memory as L1 cache

Reconfigure the partitioning of on-chip memory between L1 and shared memory:

```
cudaError_t cudaThreadSetCacheConfig(
    enum cudaFuncCache cacheConfig);

    •  cudaFuncCachePreferNone (default)
    •  cudaFuncCachePreferShared
    •  cudaFuncCachePreferL1
```



Hands On – On-Chip Memory Configuration

How does `cudaThreadSetCacheConfig` affect the performance of `iso2d`?

Try changing the call to it at the start of the `08_iso2d_cuda_aligned` example solution (`iso2d_aligned_solution.cu`) to be either:

```
cudaThreadSetCacheConfig(cudaFuncCachePreferShared);

cudaThreadSetCacheConfig(cudaFuncCachePreferL1);
```

Then, measure any change in overall performance and L1 hit ratios.

Hands On – On-Chip Memory Configuration

Version	Performance (Mcells/s/step)	Improvement
Sequential	73.35	
OpenMP	275.20	3.75
CUDA	2,708.77	9.84
+ cmem	2,812.53	1.04
+ aligned	2944.26	1.05
+ cudaFuncCachePreferShared	2943.11	1.00
+ cudaFuncCachePreferL1	3037.05	1.03

```
-x 4096 -y 4096 -i 1000
```

w/ cudaFuncCachePreferNone:	64.03% L1 Global Hit Rate
w/ cudaFuncCachePreferShared:	64.04% L1 Global Hit Rate
w/ cudaFuncCachePreferL1:	77.29% L1 Global Hit Rate

Hands On – On-Chip Memory Configuration

Version	Performance (Mcells/s/step)	Improvement
Sequential	73.35	
OpenMP	275.20	3.75
CUDA	2,708.77	9.84
+ cmem	2,812.53	1.04
+ aligned	2944.26	1.05
+ cudaFuncCachePreferShared	2943.11	1.00
+ cudaFuncCachePreferL1	3037.05	1.03

Appears to be improvement possible by increasing on-chip locality.

Time to look in to shared memory.

```
-x 4096 -y 4096 -i 1000
```

w/ cudaFuncCachePreferNone:	64.03% L1 Global Hit Rate
w/ cudaFuncCachePreferShared:	64.04% L1 Global Hit Rate
w/ cudaFuncCachePreferL1:	77.29% L1 Global Hit Rate

CUDA Shared Memory

Shared memory is shared by threads in the same block.

- On-SM memory

Shared memory can be allocated statically or dynamically

Statically Allocated Shared Memory

- Size is fixed at compile-time
- Can declare many statically allocated shared memory variables
- Can be declared globally or inside a device function
- Can be multi-dimensional

```
__shared__ int s_arr[256][256];
```

CUDA Shared Memory

Dynamically Allocated Shared Memory

- Size in bytes is set at kernel launch with a third kernel launch configurable
- Can only have one dynamically allocated shared memory array per kernel
- Must be one-dimensional array

```
__global__ void kernel(...) {
    extern __shared__ int s_arr[];
    ...
}

kernel<<<nblocks, threads_per_block, shared_memory_bytes>>>(...);
```

CUDA Shared Memory

What is reused in `iso2d`? i.e. what can benefit from improved on-chip locality?

```
for (int d = 1; d <= radius; d++) {
        int y_pos_offset = POINT_OFFSET(x, y + d, dimx, radius);
        int y_neg_offset = POINT_OFFSET(x, y - d, dimx, radius);
        int x_pos_offset = POINT_OFFSET(x + d, y, dimx, radius);
        int x_neg_offset = POINT_OFFSET(x - d, y, dimx, radius);
        div += const_c_coeff[d] * (curr[y_pos_offset] +
                curr[y_neg_offset] + curr[x_pos_offset] +
                curr[x_neg_offset]);
  }
```

Hands On – Shared Memory

Take a look at using shared memory to tile `curr`. This is the toughest transformation we'll try today.

There is also a template and example solution under `09_iso2d_cuda_smem/`.

You can start by declaring a dynamic shared memory allocation in the kernel, and figuring out how large it needs to be from the launch:

```
extern __shared__ TYPE cache[];
fwd_kernel<<<..., SHAREDY(conf.radius) * SHAREDX(conf.radius) *
        sizeof(TYPE)>>>(...);
```

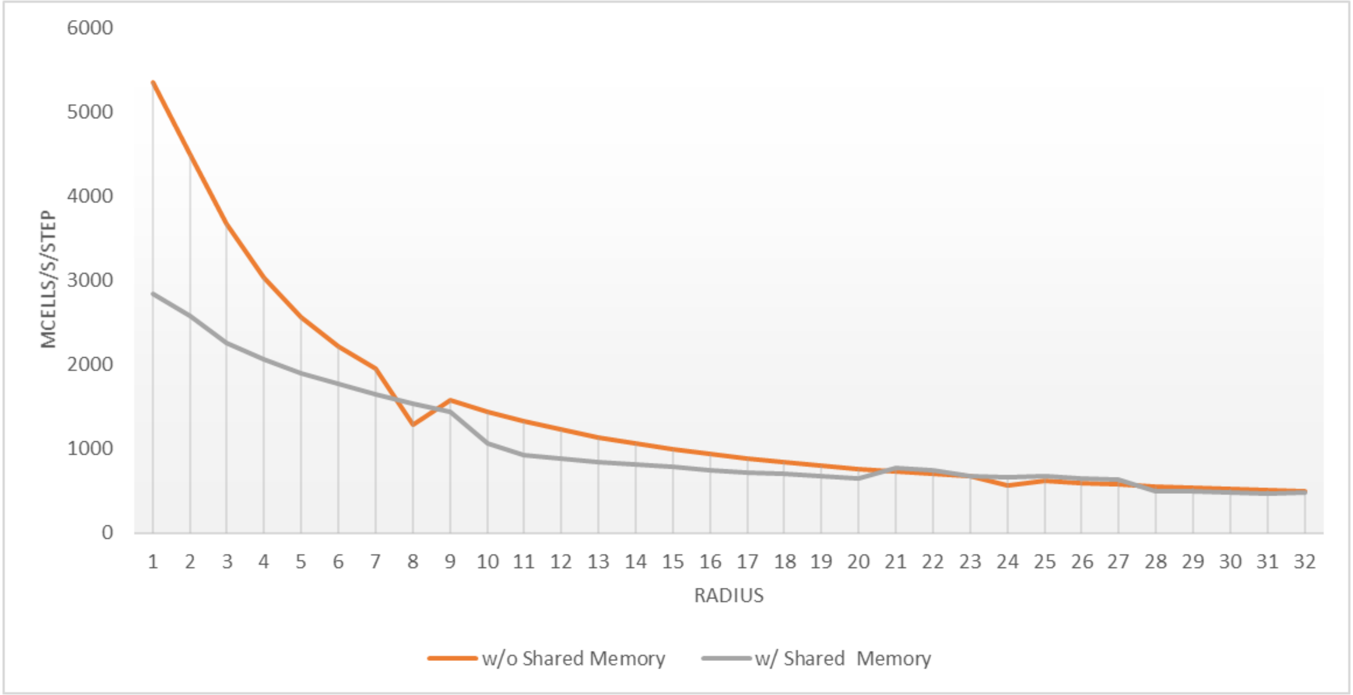
Give it a shot!

Hands On – Shared Memory

Version	Performance (Mcells/s/step)	Improvement
Sequential	73.35	
OpenMP	275.20	3.75
CUDA	2,708.77	9.84
+ cmem	2,812.53	1.04
+ aligned	2944.26	1.05
+ cudaFuncCachePreferL1	3037.05	1.03
+ smem	2063.41	0.68

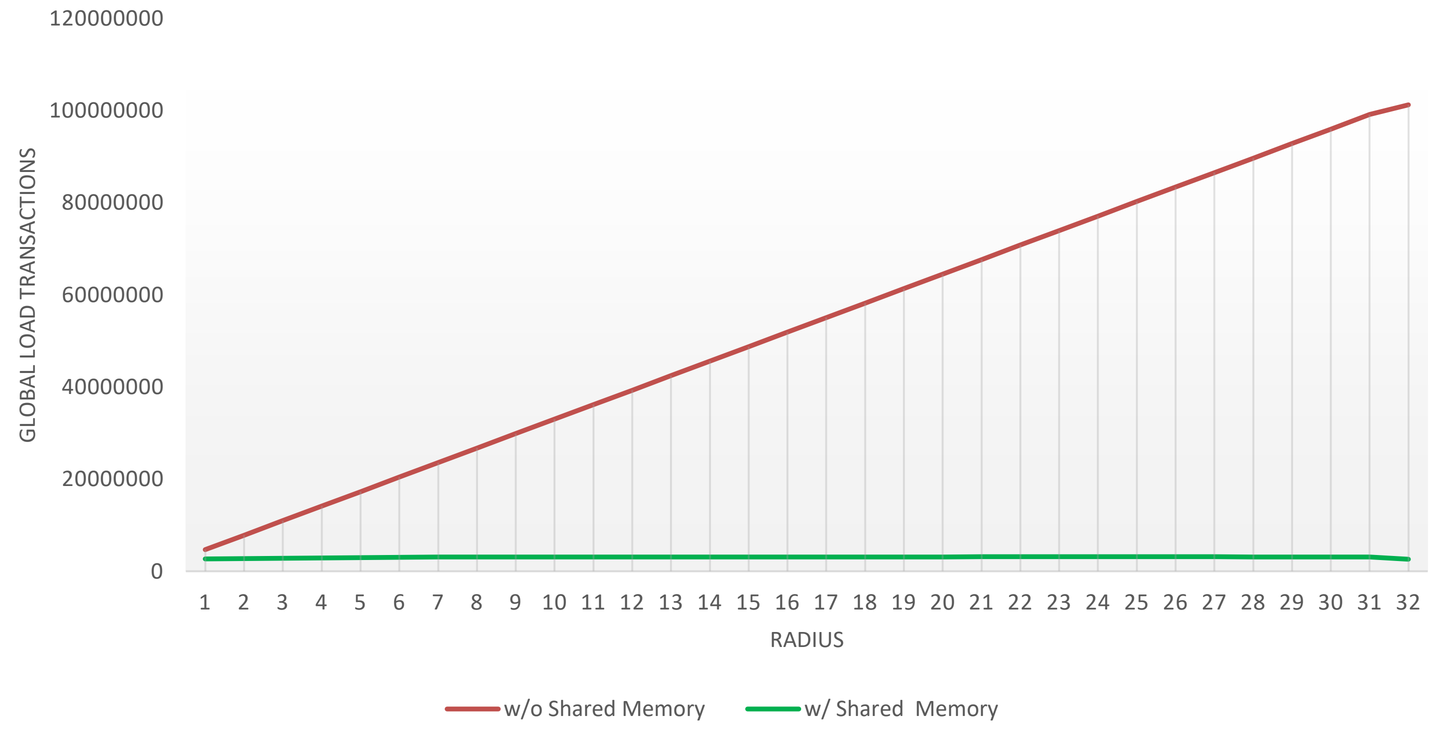
Hands On – Shared Memory

Increasing radius causes more reuse, more compute to offset overheads of cache initialization => w/ shared mem is better relative to w/o shared mem.



Hands On – Shared Memory

Very effective at reducing global load transactions.



Hands On – Shared Memory

Still don’t see any overall performance benefit because our manual management of shared memory is doing no better than the automatic management of L1, but incurs additional overheads (instructions executed, registers, etc).

Without shared memory, L1 Global Hit Rate = 77.29%

With shared memory, L1 Global Hit Rate = 0.13%

iso2d Summary

Version	Performance (Mcells/s/step)	Improvement
Sequential	73.35	
OpenMP	275.20	3.75
CUDA	2,708.77	9.84
+ cmem	2,812.53	1.04
+ aligned	2944.26	1.05
+ cudaFuncCachePreferL1	3037.05	1.03
+ smem	2063.41	0.68

Alternatives to CUDA

OpenCL

A more open standard for parallel programming of a variety of processors.

- Supports GPUs, CPUs, FPGAs

Much of the OpenCL API was based on the CUDA APIs:

CUDA API	OpenCL API
`cudaMalloc`	`clCreateBuffer`
`cudaFree`	`clReleaseMemObject`
`cudaMemcpyAsync`	`clEnqueueWriteBuffer`
`cudaMemcpyAsync`	`clEnqueReadBuffer`
`kernel<<<...>>>(...);`	`clEnqueueNDRangeKernel`

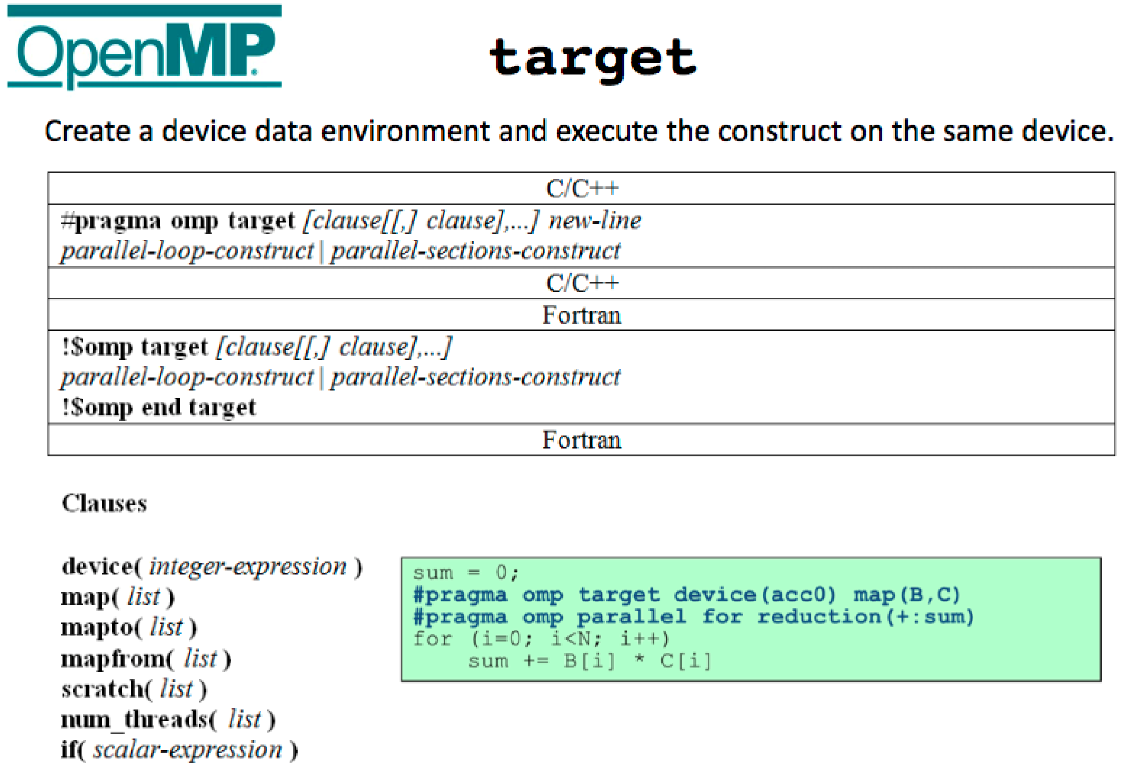
https://www.khronos.org/opencl/

OpenCL

Pros	Cons
Portable API, run one kernel everywhere.	More verbose, explicit API
Open source	**Weaker tooling than CUDA**
Not tied to a single vendor	Smaller community than CUDA, less established
Collaboration among many companies, constantly pushing the standard forward.	

https://www.khronos.org/opencl/

OpenMP Accelerators



OpenMP **target**

Create a device data environment and execute the construct on the same device.

C/C++
#pragma omp target *[clause[[,] clause],...] new-line* *parallel-loop-construct* | *parallel-sections-construct*
C/C++
Fortran
!$omp target** *[clause[[,] clause],...]* *parallel-loop-construct* \| *parallel-sections-construct* **!$omp end target
Fortran

Clauses

device(*integer-expression* **)**
map(*list* **)**
mapto(*list* **)**
mapfrom(*list* **)**
scratch(*list* **)**
num_threads(*list* **)**
if(*scalar-expression* **)**

```
sum = 0;
#pragma omp target device(acc0) map(B,C)
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++)
    sum += B[i] * C[i]
```

http://openmp.org/wp/presos/SC12/Acc%20model%20for%20OpenMP%20BOF%20SC12%20v2.pdf

OpenMP Accelerators

Pros	Cons
High-level, easy to work with	Relatively new, implementations are still coming along, community is still growing, tooling is nonexistent
Supported by a large open-source consortium	Higher abstractions mean less control over optimizations
Single parallel programming model for host and device	

http://openmp.org/wp/presos/SC12/Acc%20model%20for%20OpenMP%20BOF%20SC12%20v2.pdf

OpenACC

Pragma-based (ala OpenMP) programming for GPUs, primarily supported by PGI (a.k.a. NVIDIA).

```
#pragma acc parallel loop gang deviceptr(d_A, d_B, d_C)
    for (i = 0; i < M; i++)
#pragma acc loop worker vector
        for (j = 0; j < P; j++) {
            float sum = 0.0f;

            for (k = 0; k < N; k++)
                sum += d_A[i * N + k] * d_B[k * P + j];

            d_C[i * P + j] = sum;
        }
```

http://www.openacc.org/

OpenACC

Pros	Cons
High-level, easy to work with	With OpenMP 4.0, future of OpenACC is unclear
Earliest example of pragma-based GPU programming, so likely the most stable	Higher abstractions mean less control over optimizations
Supported by NVIDIA/PGI	

http://www.openacc.org/

Trilinos Project

"The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems."

Over 50 packages: https://trilinos.org/packages/

- Includes packages for BLAS, Preconditioners, Linear Solvers, Nonlinear Solvers, Eigensolvers, Automatic Differentiation, Domain Decomposition, Partitioning, Mesh Manipulation, …

Supports CUDA, OpenMP as backends

Provided free and open source by Sandia National Lab

https://trilinos.org/

Trilinos Project

Pros	Cons
Like a super CUDA library	Long compile times due to heavy use of C++ templating
Open source and free, developed/supported by the national labs	Performance can be hit or miss

https://trilinos.org/

Kokkos, Raja

- **Kokkos** (Sandia), **Raja** (Livermore)
  - C++ APIs that expose high-level parallel operators (e.g. map, reduce, parallel-for) and use template meta-programming to map them to multiple architectures
  - Emphasize that performance bottlenecks are predominantly in memory accesses
  - Kokkos sits under Trilinos library for computational science, used in production in industry and gov't

```
Kokkos::parallel_for(numberOfAtoms,
  [=] (const size_t atomIndex) {
    atomForces[atomIndex] =
        calculateForce(data);
  }
);
```

Example of parallel-for in Kokkos.
Blue shows Kokkos calls, red shows user-written kernel.

```
double totalIntegral = 0;
Kokkos::parallel_reduce(numIntervals,
  [=] (const size_t I,
       double & valueToUpdate) {
    valueToUpdate += function (...);
  },
  totalIntegral);
```

Example of parallel reduce in Kokkos.
Blue shows Kokkos calls, red shows user-written kernel

GPU Acceleration of the JVM

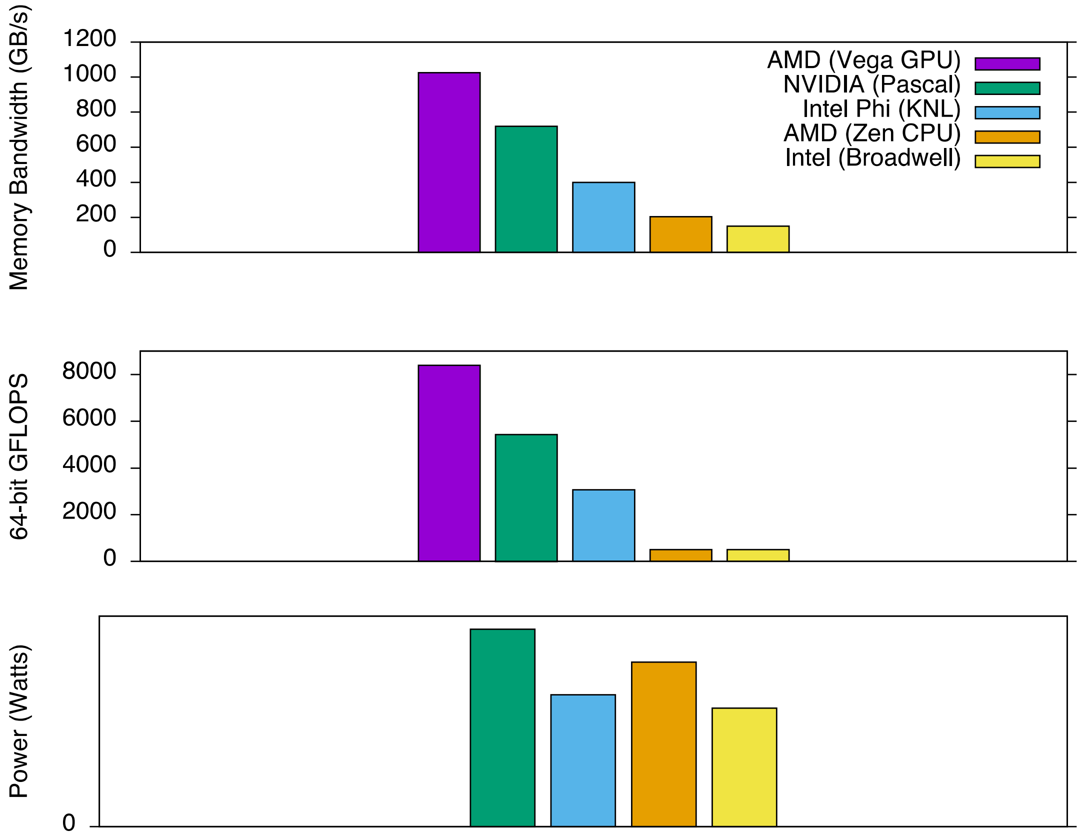
Many projects looking at JVM acceleration:

- APARAPI: https://github.com/aparapi/aparapi
- Rootbeer: https://github.com/pcpratts/rootbeer1
- SWAT: https://github.com/agrippa/spark-swat
- IBM's J9 JVM: http://ahayashi.blogs.rice.edu/files/2013/07/IBM_Java8_GPU_PACT15_cameraready-sj0tik.pdf

Pros	Cons
Broader set of GPU applications	Overheads from JIT-ing, serialization, etc may make speedup harder to achieve
High-level JVM programming languages	Most solutions are still in the research stage
Speedup relative to JVM-based apps may be > speedup relative to native apps	Little or no control over optimization

Alternatives to GPUs

FPGA, KNL, x86



Is your hardware holding you back, or are you holding back your hardware?

GPUs have higher hardware peaks, but also have performance characteristics that you <u>actually stand a chance of reasoning about</u>.

Summary

Topics Not Covered

Multi-GPU CUDA Programming

CUDA-Aware MPI

Advanced Stream Usage

Advanced Instruction Optimization

Unified Memory

Texture Memory

Shared Bank Conflicts

CUDA libraries

Dynamic Parallelism

Warp Shuffle Instructions

`cuda-gdb`

…

Topics Covered

GPU Architecture

nvprof

CUDA Execution Model

Kernel Synchronization

CUDA Asynchrony

CUDA Memory Hierarchy

Optimizing CUDA Data Access

Hands On with a Non-Trivial CUDA Application

CUDA Atomic Operations

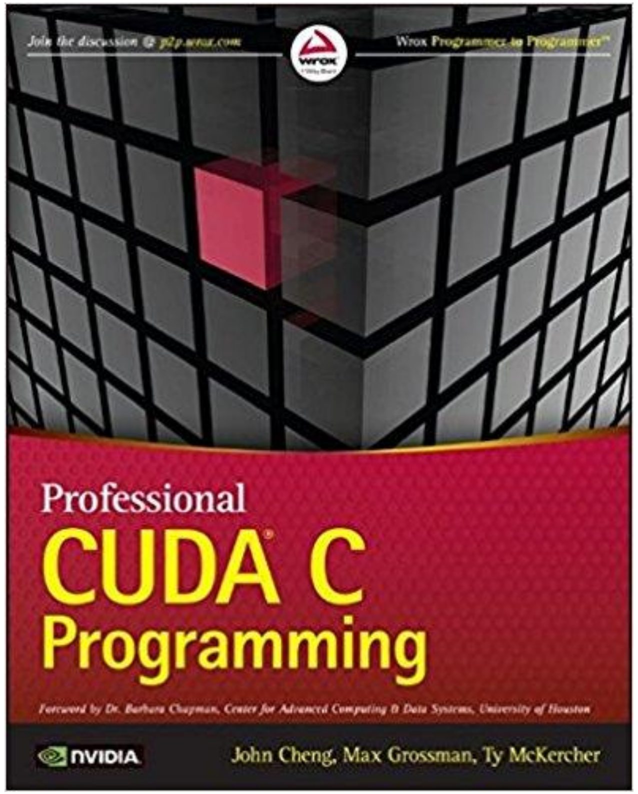
Alternatives to CUDA

Additional Resources

I'm not condoning theft, but…

If you happen to google "Professional CUDA C Programming PDF", the first result may or may not be a bootleg copy of my textbook, on which these slides are based.

Contact:
Max Grossman
max@7pod.tech


Join the discussion @ p2p.wrox.com
wrox
Wrox Programmer to Programmer™
Professional
CUDA® C
Programming
Foreword by Dr. Barbara Chapman, Center for Advanced Computing & Data Systems, University of Houston
NVIDIA
John Cheng, Max Grossman, Ty McKercher