

EN 600.320/420 Parallel Programming

CUDA C Programming

What Is CUDA?

- **CUDA**: Compute Unified Device Architecture
- Created by NVIDIA
- A way to perform computation on the GPU
- Specification for:
 - A computer architecture
 - A language
 - An application interface (API)

CUDA Execution Model

- A CUDA device is a highly parallel processor
- We assume it can execute many hundreds of threads in parallel
- Threads to Stream Processors ratio > 1
- When writing CUDA software, think in terms of threads, not processors
- Startup and Context Switching costs per thread are very low!!

CUDA Execution Model

- The CUDA programming model imposes a data decomposition approach
- The *grid* is the data domain (1D, 2D or 3D)
- The grid is decomposed into *thread blocks*
- A thread block is decomposed into *threads*

CUDA Data Decomposition

Device

Grid

B (0,0)	B (0,1)
B (1,0)	B (1,1)
B (2,0)	B (2,1)
...	

Thread Block (1,1)

T (0,0)	T (0,1)
T (1,0)	T (1,1)
T (2,0)	T (2,1)
...	

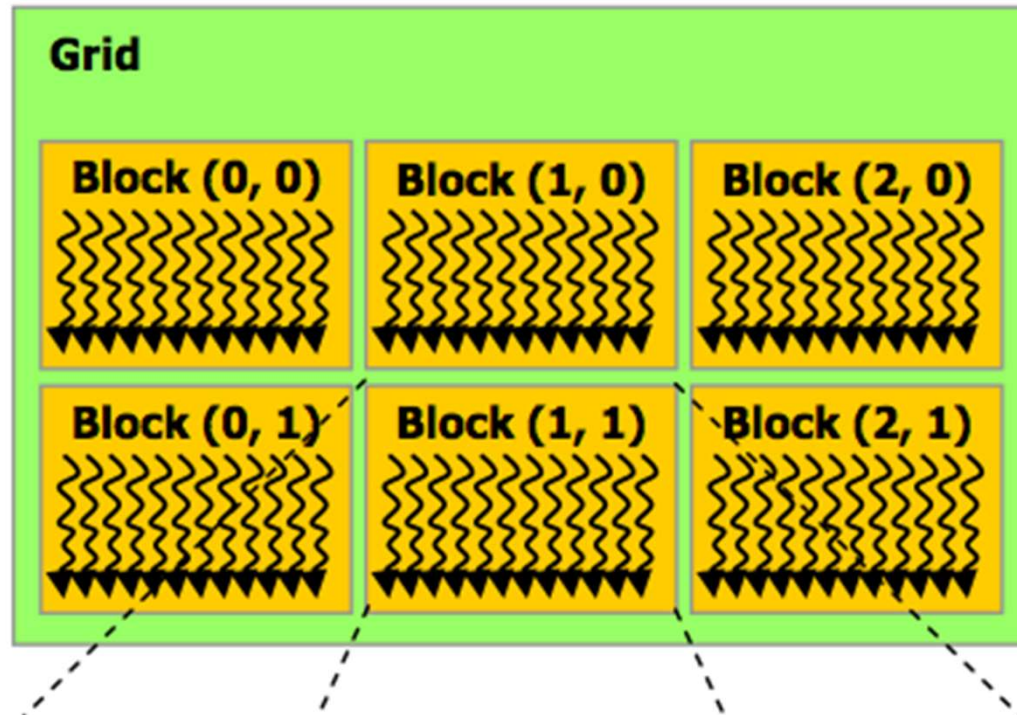
CUDA Execution Model

- Thread blocks and threads are given unique identifiers
- Identifiers can be 1D, 2D or 3D
- Used by the kernel to identify which part of a problem to work on
 - E.g. which data from memory to read, etc.

CUDA Kernel

- A *kernel* is a program that processes a single data element
- A thread runs the kernel on a data element

Grid->Blocks->Threads

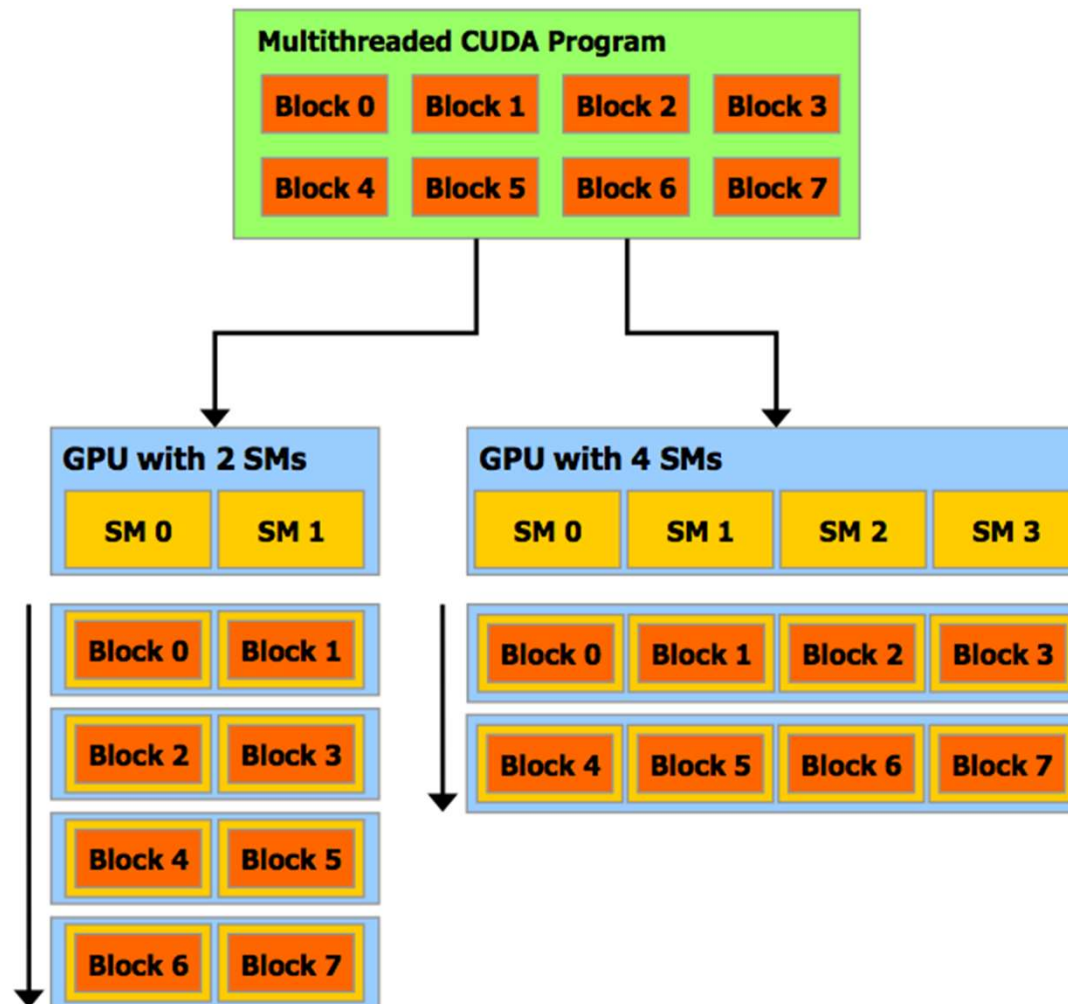


CUDA Execution Model

Thread Blocks

- A thread block may have up to 512 threads
- All threads in a thread block are run on the same multi-processor
 - Thus can communicate via shared memory
 - And synchronize
- Threads of a block are multiplexed onto a multi-processor as *warps*

Thread Block Scheduling

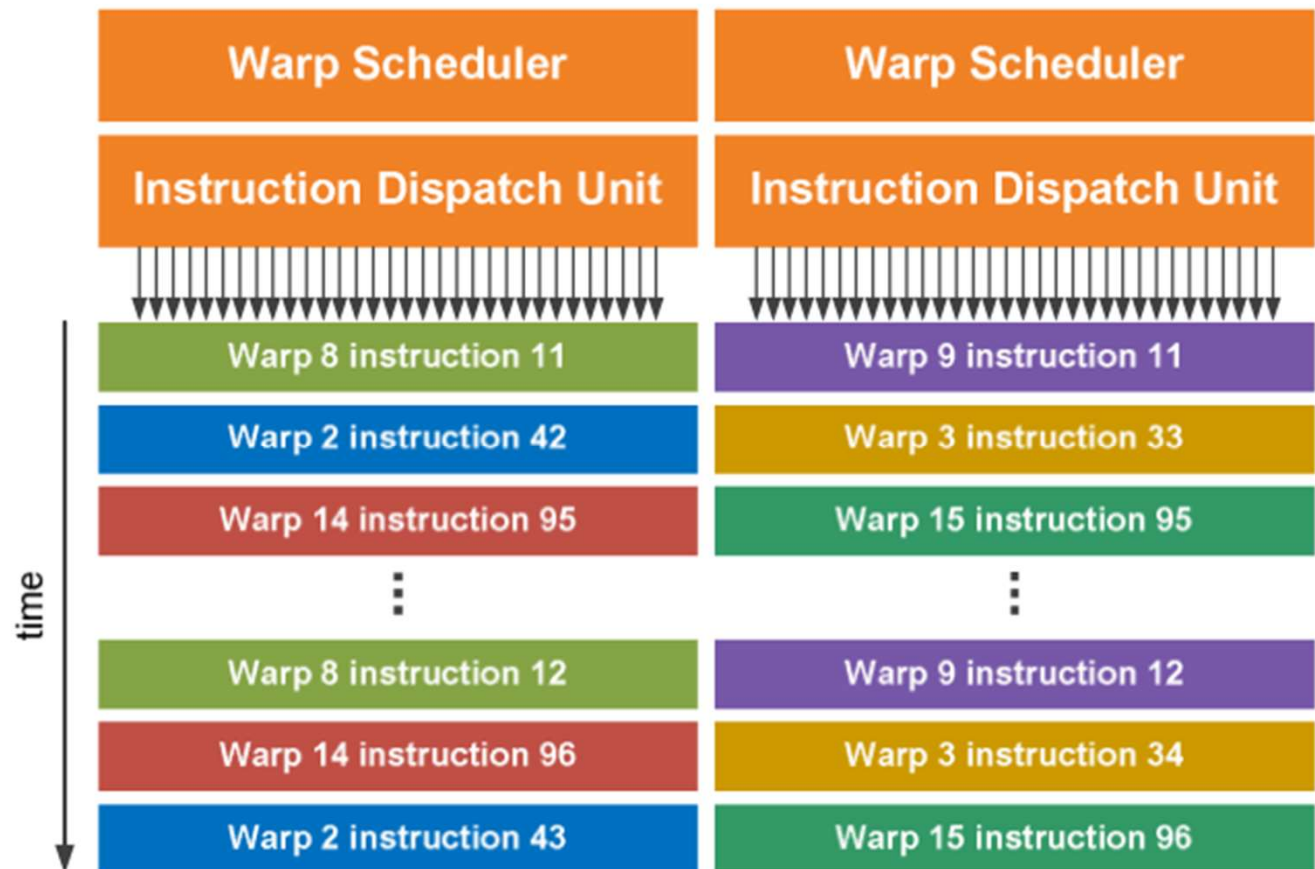


Warps

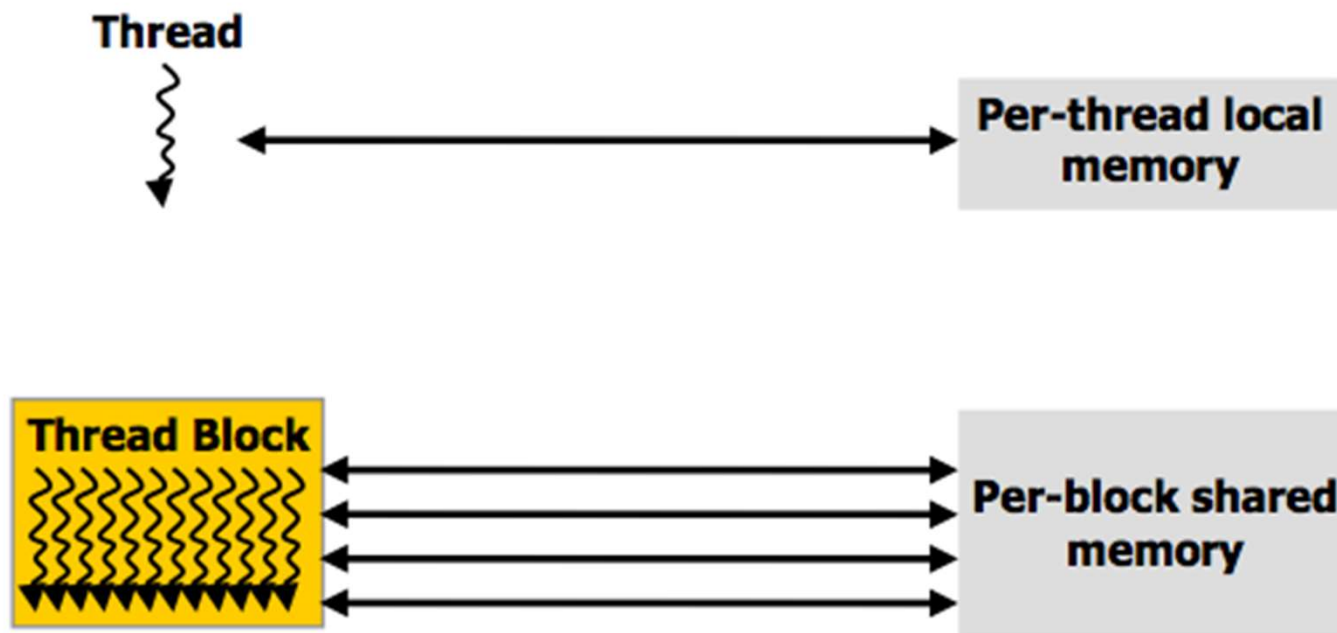
- Warps are groups of 32 threads
- Warps are the fundamental scheduling unit of the processor
 - Dispatched two at a time to 16 processors each (on Fermi)
- Each warp forms a SIMD group
 - Thread blocks are not SIMD, Warps are!

Warp Scheduling

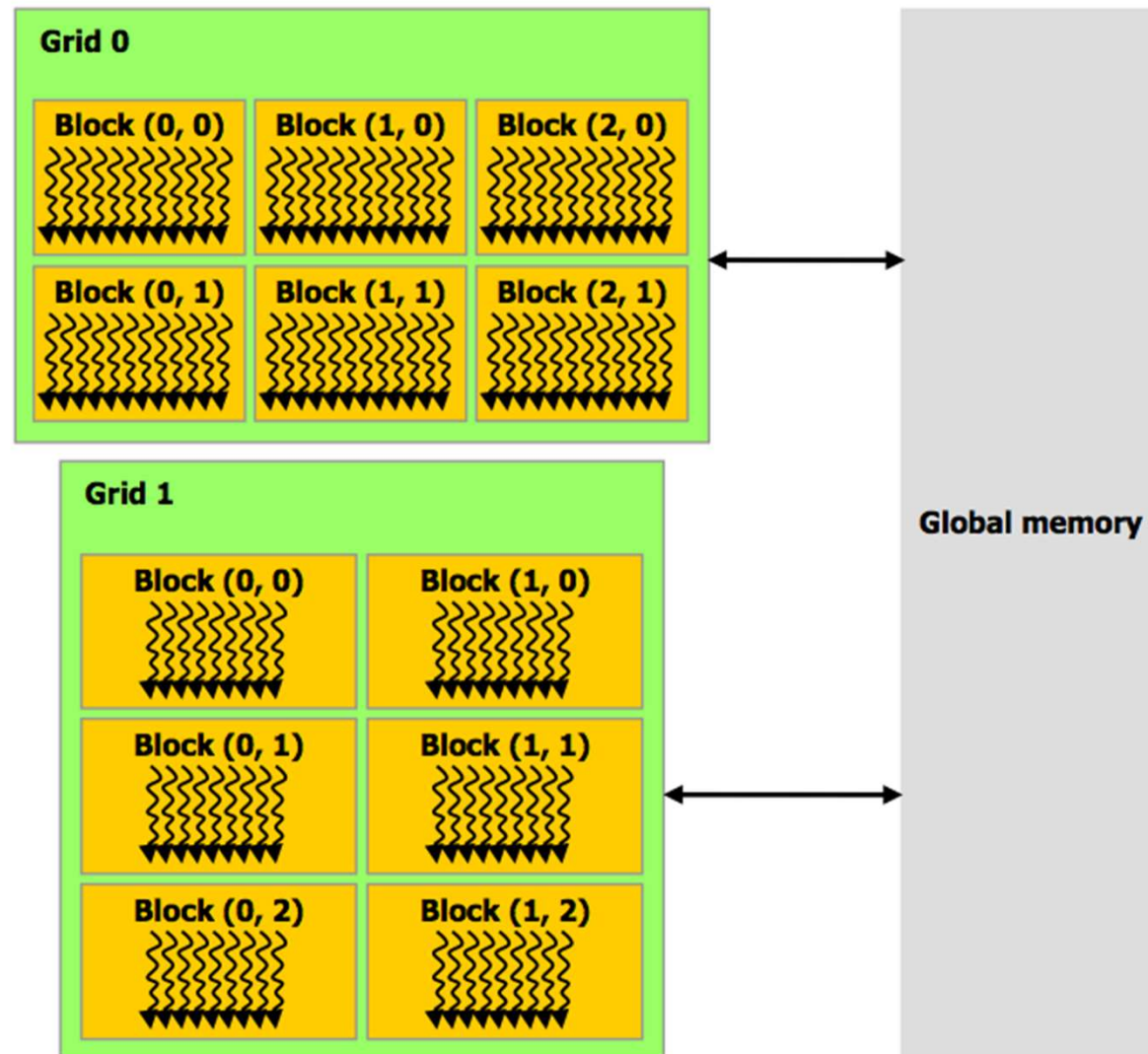
- Dual issue instructions
 - 32 threads
 - 16 cores
- 2 units/SM



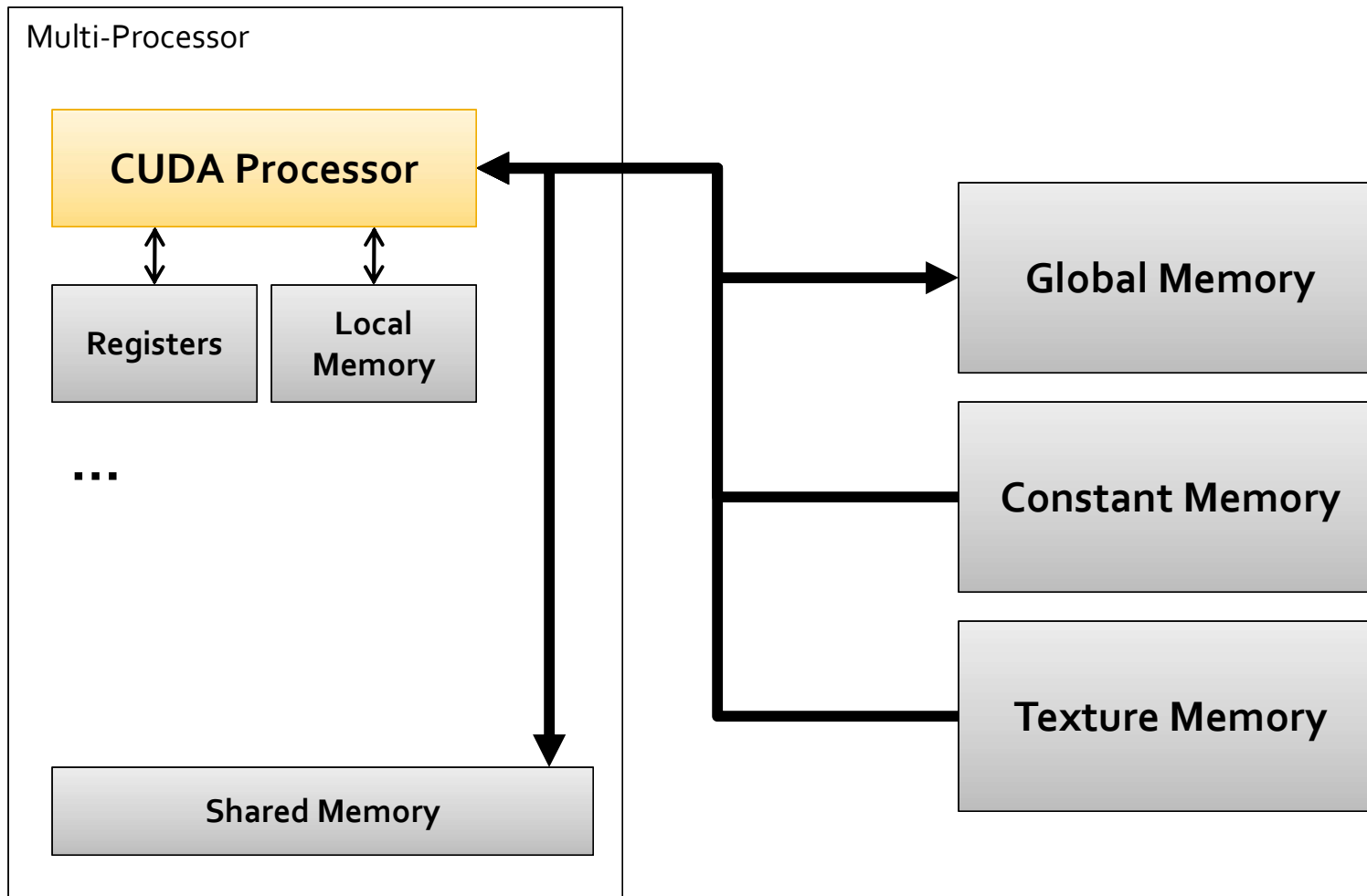
Memory Addressability



Memory Addressability



CUDA Memory Hierarchy



CUDA Memory Hierarchy

CUDA Processors have access to:

Memory Type	Access	Sharing
Registers	Read/Write	Private
Local Memory	Read/Write	Private
Shared Memory	Read/Write	Multi-Processor
Global Memory	Read/Write	Device
Constant Memory	Read	Device
Texture Memory	Read	Device

CUDA Memory Model

Registers

- Large number of registers per stream processor (1024)
- Zero-clock cycle access
- Store either 64 bit integer or 64 bit float

CUDA Memory Model

Shared Memory

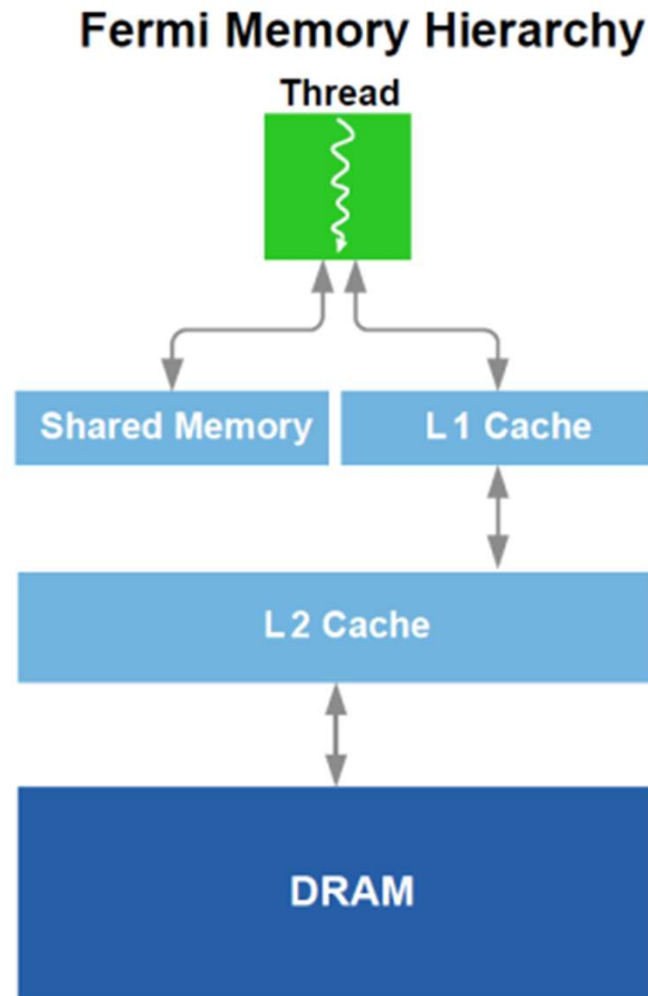
- A block of memory that is shared by all stream processors in a multi-processor
 - 48 or 16K per SM in Fermi
- 16KB per block, stored in 16x1KB banks
- Very fast to access (i.e. as fast as registers!) without *bank conflicts*

CUDA Memory Model

Global Memory

- The large block of memory shared by all multi-processors on the compute device
- Size depends on device – 256MB to 24GB (Tesla K80 is 2x 12GB)
- High bandwidth (K80 is 480 GB/s)
- Slow to access – several hundred clock cycle latency.

Fermi Memory Model

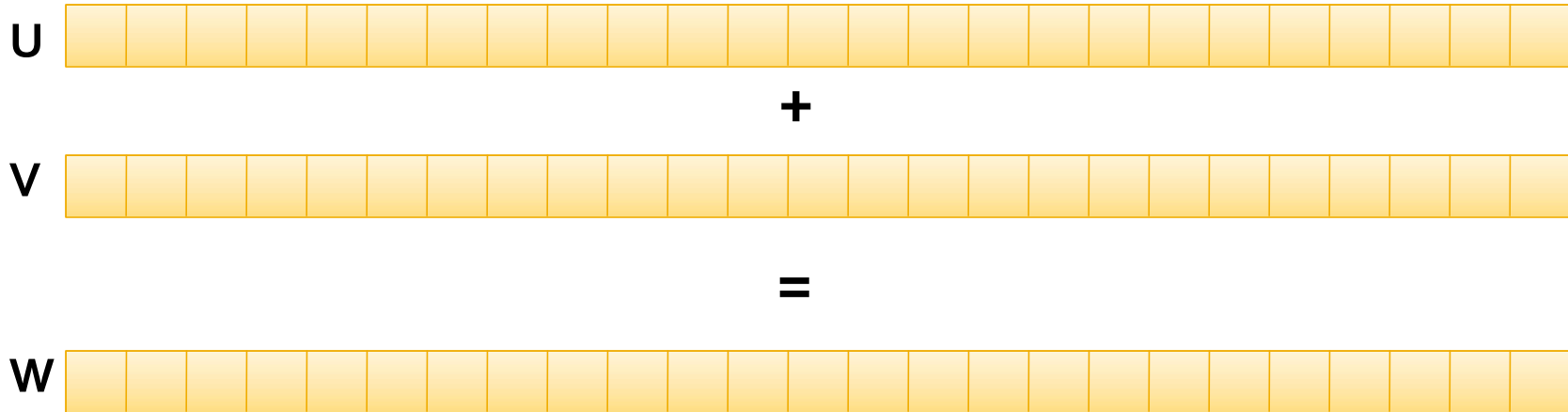


First CUDA Program

CUDA Program

- We will create a simple CUDA program to add two vectors
- $U = \{u_0, u_1, \dots, u_n\}$
- $V = \{v_0, v_1, \dots, v_n\}$
- $W = U + V = \{u_0 + v_0, u_1 + v_1, \dots, u_n + v_n\}$

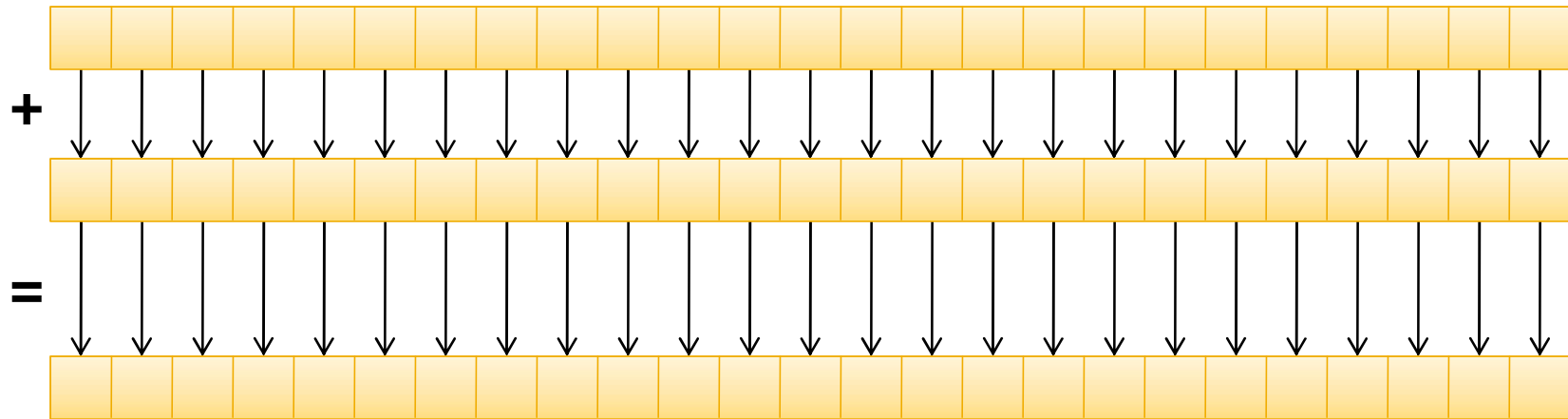
Vector-Vector Addition



$$W = U + V \quad W_i = U_i + V_i$$

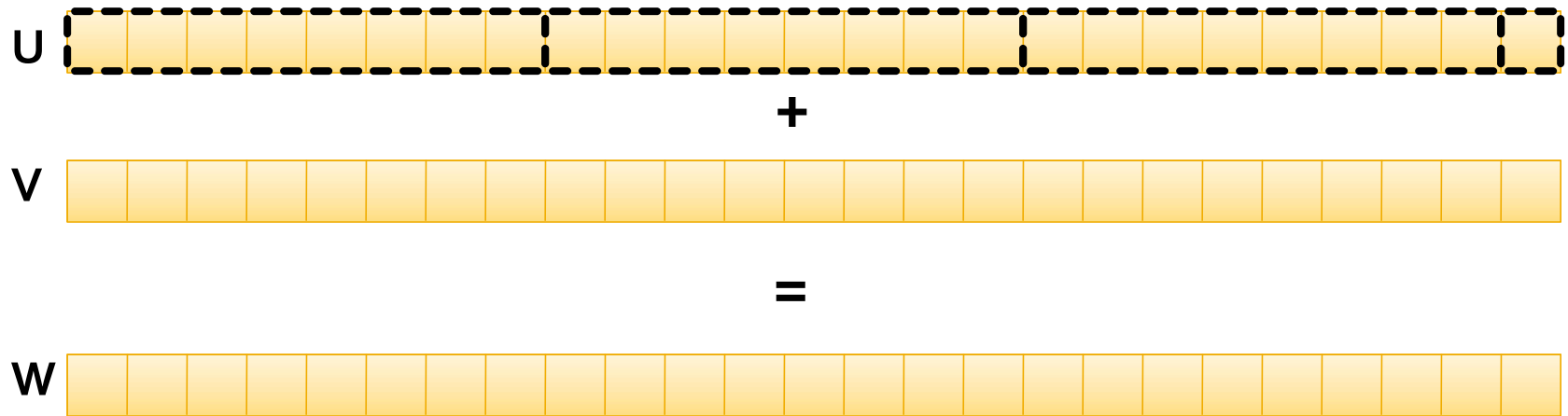
- Easy to parallelize: Each element is independent!

Threads



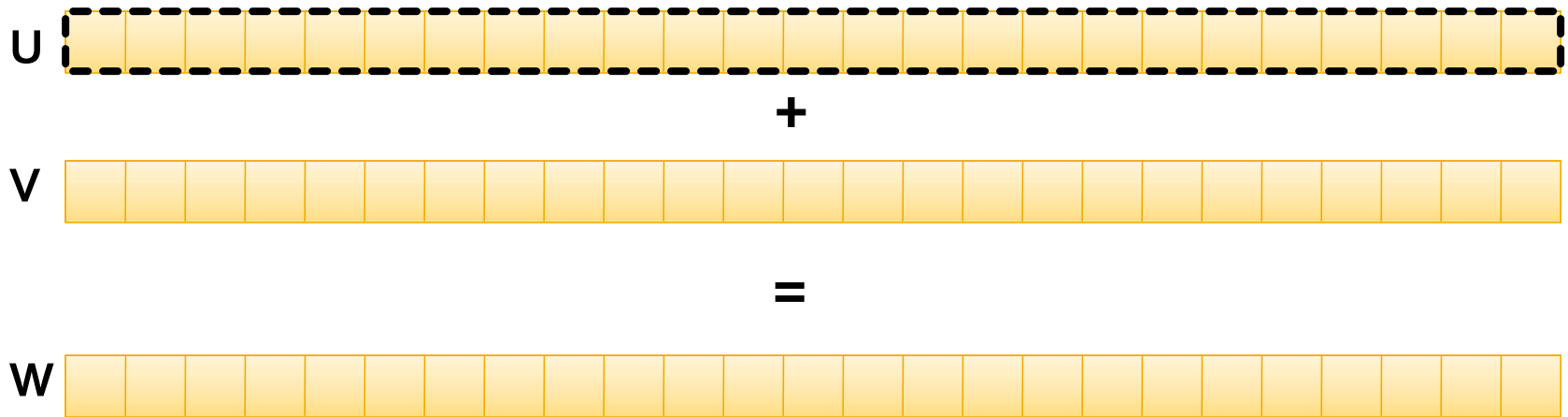
- **Threads:** Each element is computed by a separate thread

Thread Blocks



- **Blocks:** Group sets of adjacent elements into blocks
 - To conform with device parameters

Grid

$$\begin{array}{l} \mathbf{u} \\ + \\ \mathbf{v} \\ = \\ \mathbf{w} \end{array}$$


- **Grid:** The entire vector

Device Code

- Our program will be a single source file, with two parts:

Device Code

- A kernel to perform the addition of two elements

Kernel Function

```
__global__ void VectorAdditionKernel(  
    const float* pVectorA,  
    const float* pVectorB,  
    float* pVectorC)  
{  
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;  
    pVectorC[i] = pVectorA[i] + pVectorB[i];  
}
```

CUDA Language

- CUDA defines a language that is similar to C/C++
- CUDA source code contains a mix of *host* and *device* code and data

CUDA Language

- CUDA defines a language that is similar to C/C++
- Important Differences:
 - Runtime Library
 - Functions
 - Classes, Structs, Unions

CUDA Runtime Library

- Code that runs on the device can't use normal C/C++ Runtime Library functions
- No `printf`, `fread`, `malloc`, etc
- Most math functions have device equivalent

CUDA Runtime Functions

- There are a number of device specific functions/intrinsics available:
 - `__syncThreads`
 - `__mul24`
 - `atomicAdd`, `atomicCAS`, `atomicMin`, ...

Functions

- On a CUDA device, there is no stack
- By default, all function calls are inlined
- All local variables, function arguments are stored in registers
- **NO** function recursion
- No function pointers

CUDA Language

- CUDA defines a language that is similar to C/C++
- Syntactic extensions:
 - Declaration Qualifiers
 - Built-in Variables
 - Built-in Types
 - Execution Configuration

Declspec's

- Declspec = declaration specifier / declaration qualifier
- A modifier applied to declarations of:
 - Variables
 - Functions
- Examples: `const`, `extern`, `static`

CUDA Function Declspecs's

- CUDA uses the following declspecs for functions:
- `__device__`
- `__host__`
- `__global__`

`__device__`

- Declares that a function is compiled to, and executes on the device
- Callable only from another function on the device

`__host__`

- Declares that a function is compiled to and executes on the host
- Callable only from the host
- Functions without any CUDA declspec are host by default

__global__

- Declares that a function is compiled to and executes on the device
- Callable from the host
- Used as the entry point from host to device

Kernel Function

```
__global__ void VectorAdditionKernel(  
    const float* pVectorA,  
    const float* pVectorB,  
    float* pVectorC)  
{  
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;  
    pVectorC[i] = pVectorA[i] + pVectorB[i];  
}
```


Vector Types

- Can construct a vector type with special function:
`make_{typename}(v_0, v_1, ...)`
- Can access elements of a vector type with
".x", ".y", ".z", ".w": `vecvar.x`
- `dim3` is a special vector type for grids, same as `uint3`

Built-in Variables

- CUDA provides four global, built-in variables
- `threadIdx`, `blockIdx`, `blockDim`, `gridDim`
- Typed as a `'dim3'` or `'uint3'`
- Accessible only from device code
- Cannot take address
- Cannot assign value

Host Code

- Our program will be a single source file, with two parts:

Host Code

- Allocate GPU memory for vector
- Copy vector from host to device memory
- Launch kernel
- Copy vector from device to host memory

Kernel Function

```
__global__ void VectorAdditionKernel(  
    const float* pVectorA,  
    const float* pVectorB,  
    float* pVectorC)  
{  
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;  
    pVectorC[i] = pVectorA[i] + pVectorB[i];  
}
```

Host Code – Copy Data To GPU

...

```
float* pDeviceVectorA = 0;  
float* pDeviceVectorB = 0;  
float* pDeviceVectorC = 0;
```

```
cudaMalloc((void**)&pDeviceVectorA, VectorSize);  
cudaMalloc((void**)&pDeviceVectorB, VectorSize);  
cudaMalloc((void**)&pDeviceVectorC, VectorSize);
```

```
cudaMemcpy(pDeviceVectorA, pHostVectorA,  
           VectorSize, cudaMemcpyHostToDevice);  
cudaMemcpy(pDeviceVectorB, pHostVectorB,  
           VectorSize, cudaMemcpyHostToDevice);
```

...

Allocating Memory

- There is no `malloc` or `free` function that can be called from device code
→ How can we allocate memory?
- From the host with `cudaMalloc()`
 - And copy data in from host to initialize

Host Code

```
bool VectorAddition(  
    unsigned N,  
    const float* pHostVectorA,  
    const float* pHostVectorB,  
    float* pHostVectorC)  
{  
    const unsigned BLOCKSIZE = 512;  
    unsigned ThreadCount = N;  
    unsigned BlockCount = N / BLOCKSIZE;  
    unsigned VectorSize = ThreadCount * sizeof(float);  
  
    ...  
}
```

Host Code – Execute Kernel

...

```
VectorAdditionKernel<<<BlockCount,BLOCKSIZE>>>(  
    pDeviceVectorA,  
    pDeviceVectorB,  
    pDeviceVectorC);
```

...

Execution Configuration

- CUDA provides syntactic sugar to launch the execution of kernels

```
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```

Execution Configuration

```
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```

- Func is a `__global__` function

Execution Configuration

```
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```

- GridDim is a 'dim3' typed expression giving the size of the grid (i.e. problem domain)

Execution Configuration

```
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```

- BlockDim is a 'dim3' typed expression giving the size of a thread block

Execution Configuration

```
Func<<<GridDim, BlockDim>>>(Arguments, ...)
```

- The compiler turns this type of statement into a block of code that configures, and launches the kernel

Host Code – Read result from GPU

```
...  
  
    cudaMemcpy(pHostVectorC, pDeviceVectorC, VectorSize,  
               cudaMemcpyDeviceToHost);  
  
    ...  
}
```

CUDA Variable Declspec's

- CUDA uses the following declaration qualifiers for variables:
- `__device__`
- `__shared__`
- `__constant__`
- Only apply to global variables

`__device__`

- Declares that a global variable is stored on the device
- The data resides in global memory
- Has lifetime of the entire application
- Accessible to all GPU threads
- Accessible to the CPU via API

__shared__

- Declares that a global variable is stored on the device
- The data resides in shared memory
- Has lifetime of the thread block
- Accessible to all threads, one copy per thread block

`__shared__`

- If not declared as `volatile`, reads from different threads are not visible unless a synchronization barrier used
- Not accessible from CPU

__constant__

- Declares that a global variable is stored on the device
- The data resides in constant memory
- Has lifetime of entire application
- Accessible to all GPU threads (read only)
- Accessible to CPU via API (read-write)

Vector Size

- What if the vector size is not an integral number of blocks?
- **Option 1:**
Perform bounds checking in kernel
- **Option 2:**
Pad out the vector to correct length

Maximum Vector Size

- Grid is 1-dimensional
- Maximum of 512 threads in a block
- Maximum of 65536 blocks in a 1D grid
 - Maximum vector size is 65536 times 512
- How do we operate on a vector larger than 16M elements?

Maximum Vector Size

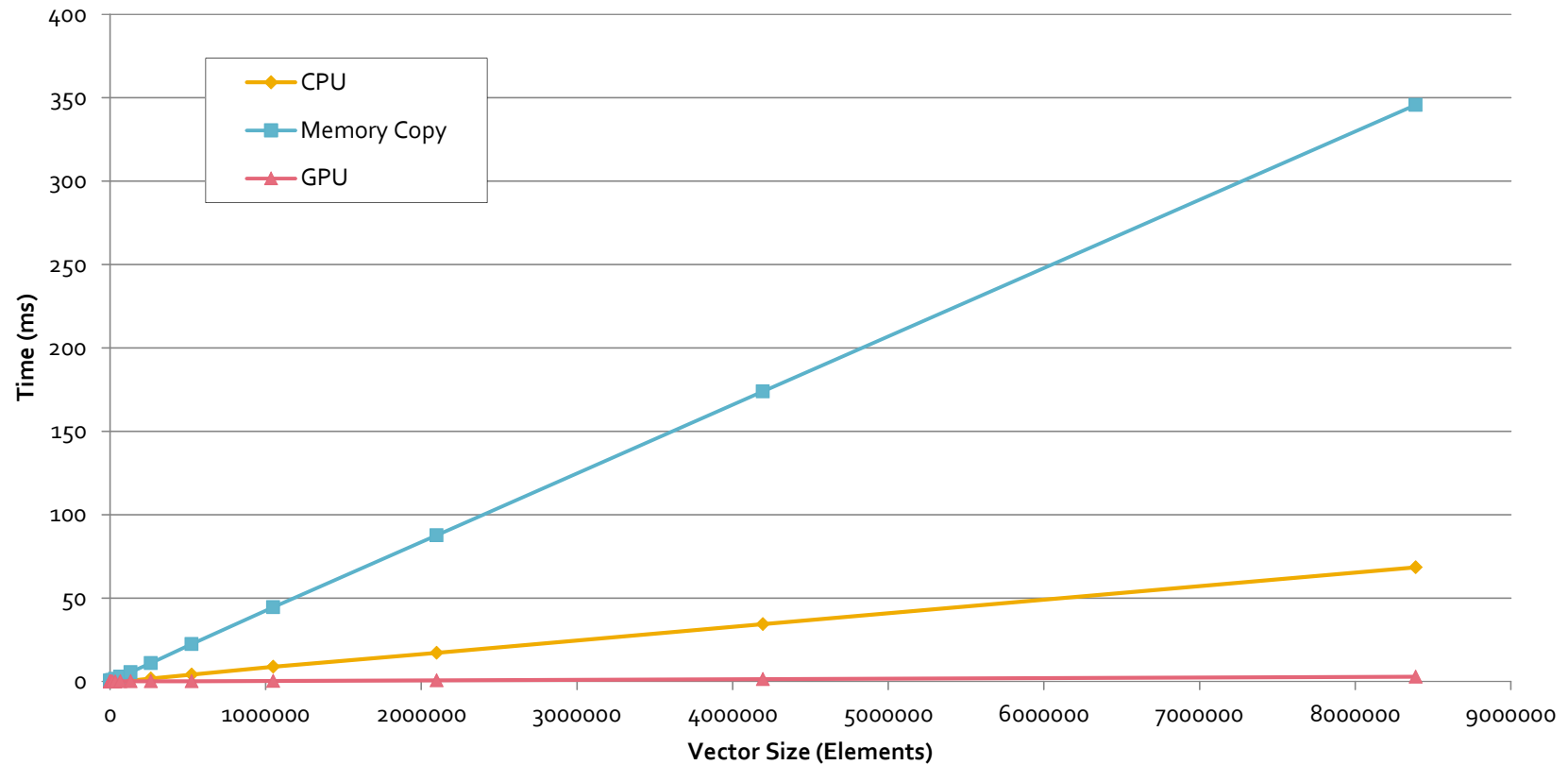
- How do we operate on a vector larger than 16M elements?
- **Option 1:**
Use a 2-D indexing scheme
- **Option 2:**
Compute with several grids

Performance

- The GPU is faster than the CPU
- But, computing on the GPU involves overhead:
 - Must get data to/from the GPU
- Where is the “break-even” point?

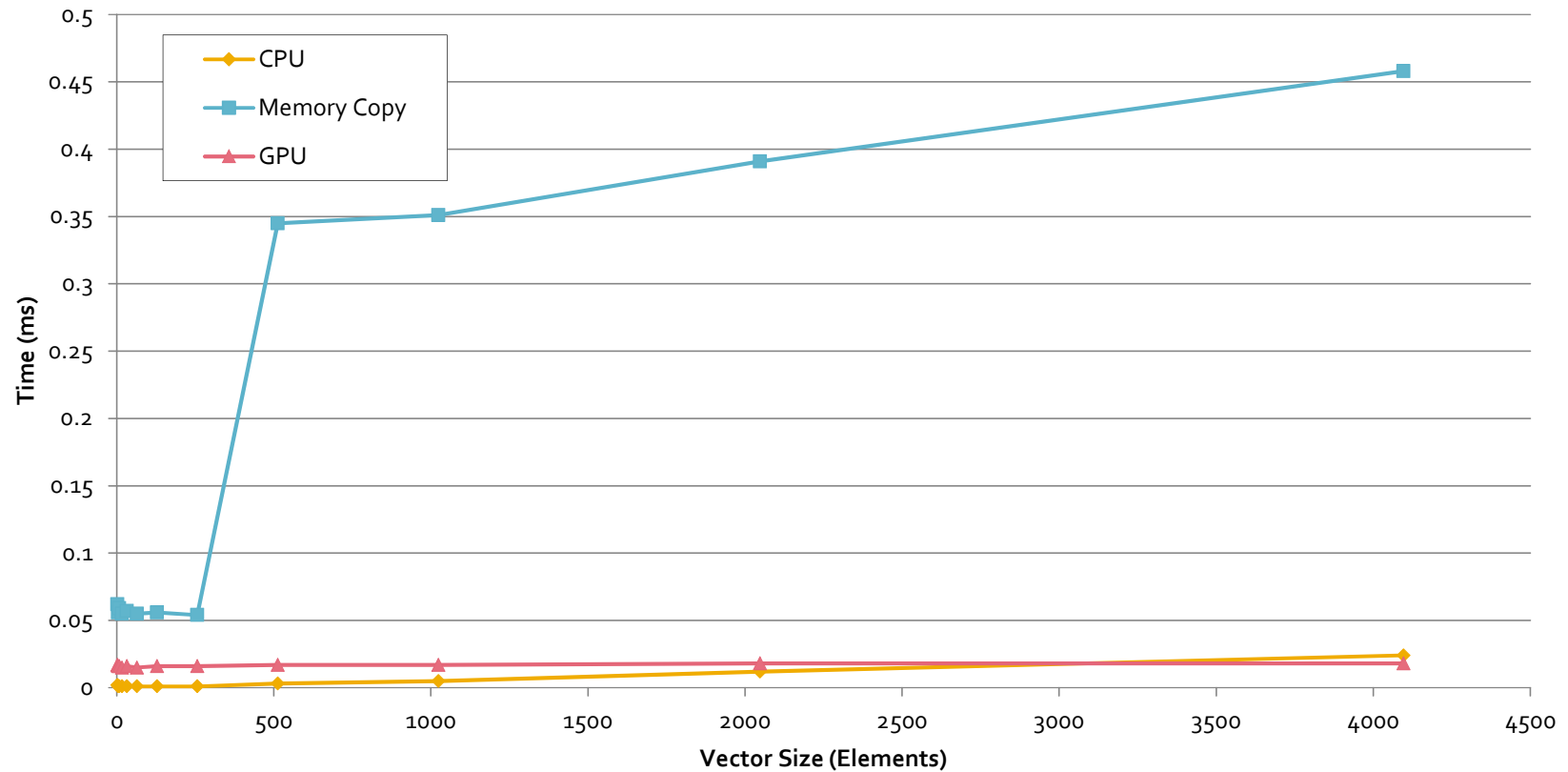
Vector Addition Performance

CUDA Vector-Vector Addition Performance



Vector Addition Performance

CUDA Vector-Vector Addition Performance



Performance Conclusion

- Matrix addition is not a good CUDA program
 - Why? In terms of Roofline? Not enough operational intensity
- Never overcome the data transfer
 - Not enough computation
 - Better on the CPU