GPU Computing Workshop:
Basic CUDA C Programming

7 Oktober 2016 – FMIPA, Universitas Andalas

Muhammad Teguh Satria, MSc

NovaGlobal Pte Ltd
Green & Scientific Computing Solutions

NVIDIA PREFERRED SOLUTION PROVIDER

Profile

- Name: Muhammad Teguh Satria (Teguh)
- Current Position: Advanced Systems Engineer at NovaGlobal Pte Ltd
- Education:
  - BSc Mathematics, Bandung Institute of Technology, Indonesia.
  - MSc Computer Science, Taipei Tech, Taiwan.
- Related Work/Research Experiences:
  - *GPU Performance Model for Control Flow Divergence,* **ADSC (SG-based Lab of University of Illinois Urbana-Champaign), Singapore.**
  - *Attentive Teleconferencing Robot on Jetson TK1 GPU,* **ADSC *(collaboration with A*STAR I2R)*, Singapore.**
  - *GPU Implementation for Atmospheric Science,* **Space Science and Engineering Center, University of Wisconsin-Madison, US.**
  - *GPU for Geothermal,* **TerraMath Indonesia**
  - *Parallel Implementation of Tsunami Simulation on GPU,* **Taipei Tech, Taiwan**.

GPU Computing

- GPU as CPU co-processor: expose GPU parallelism for general-purpose computing
- CPU has low latency but GPU offers high throughput
- Legacy GPU Computing: Shader Programming
  - Cg, developed by NVIDIA
  - GLSL, developed by Khronos Group.
  - BrookGPU, developed by Standford University
- Modern GPU Computing Platform
  - CUDA, proposed and developed by NVIDIA
  - OpenCL, proposed by Apple, developed and maintained by Khronos Group.
  - DirectCompute, developed by Microsoft, bundled in DirectX SDK.

*image source: geforce.com, freeiconspng.com, pixabay.com

Basic CUDA Concepts

Basic CUDA Concepts

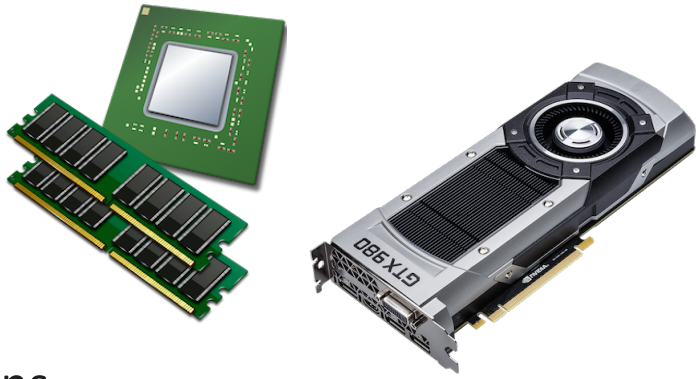
- CUDA Platform
- GPU Memory
- CUDA Kernel
- CUDA Thread
- Device Management
- Compiling & Debugging

Useful Prior Knowledge

- C/C++ experiences with:
  - Pointers and addresses
  - Memory allocation and deallocation
  - Arrays and indexing
- Multithreading concept (POSIX Thread, OpenMP, MPI)

Terminologies in CUDA Platform

- *Host*: the CPU and DRAM.



- *Device*: the GPU and its DRAM.
- *Thread*: the smallest sequence of instructions.

- *Warp: a set of 32 threads*
- *Kernel*: functions in the code that will be sent to GPU and executed in parallel by threads.
- *Memory*: refers to DRAM in CPU and GPU
- *Shared memory*: On-chip memory in GPU

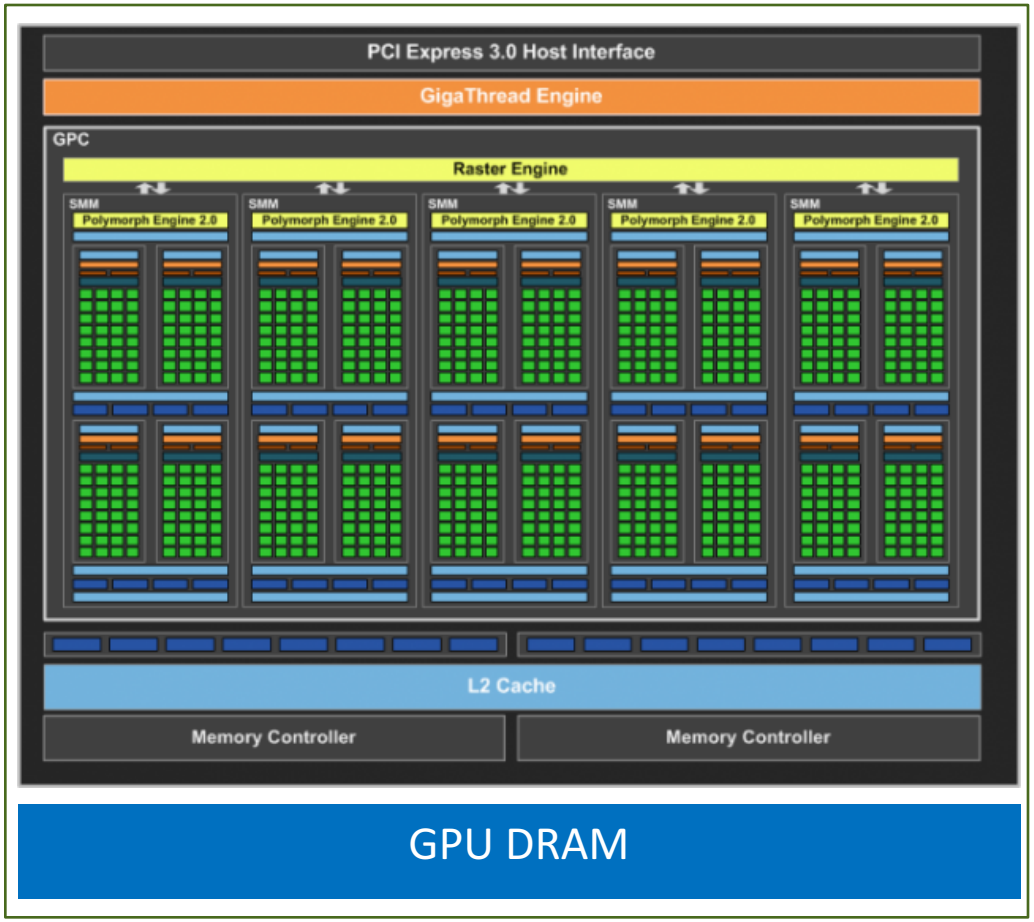
CUDA Platform

- GPU Memory
- CUDA Kernel
- CUDA Thread
- Device Management
- Compiling & Debugging

CUDA Platform

- CUDA is a computing platform that allows us to expose GPU for general purpose processing.
- CUDA is developed by NVIDIA and only works on NVIDIA GPUs.
- CUDA Platform = Architecture + API + Libraries
- CUDA Architecture: G80, Fermi, Kepler, Maxwell, Pascal
- CUDA API
  - Official supported language
    - C/C++ called as CUDA C (free)
    - Fortran, called as CUDA Fortran. NVIDIA acquired PGI in 2013. (licensed)
    - OpenACC, directive-based programming model. (licensed)
  - Developed by 3rd party:
    - JCUDA, Java binding for CUDA (open source)
    - PyCUDA, Python wrapper for CUDA (open source)
    - GPU.NET for C# (licensed)
    - Matlab, Mathematica, LabVIEW (licensed)
- CUDA Libraries: Thrust, cuBLAS, cuFFT, cuRand, etc.

CUDA Architecture – Maxwell

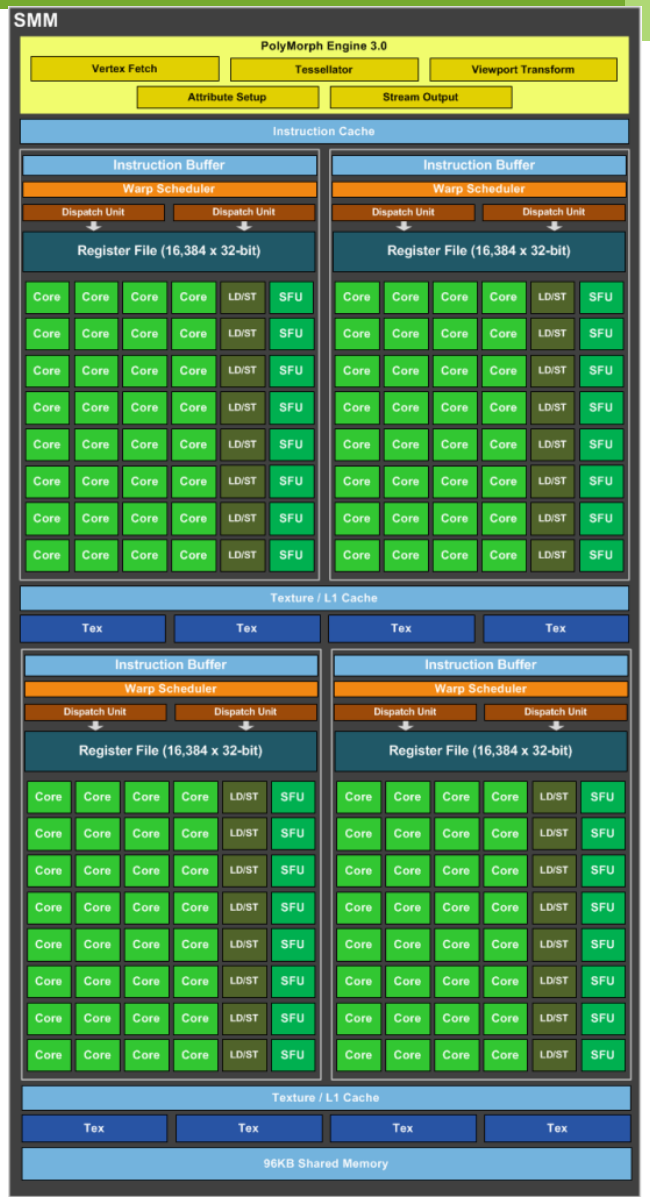


- Streaming Multiprocessor (SMM)
- L2 Cache for GPU Memory caching
- GPU DRAM (Global Memory)

CUDA Architecture – Maxwell

1 SMM has

- 128 cores
- 4 warp schedulers
- Register
- L1 cache
- Texture cache
- Constant cache
- Shared Memory



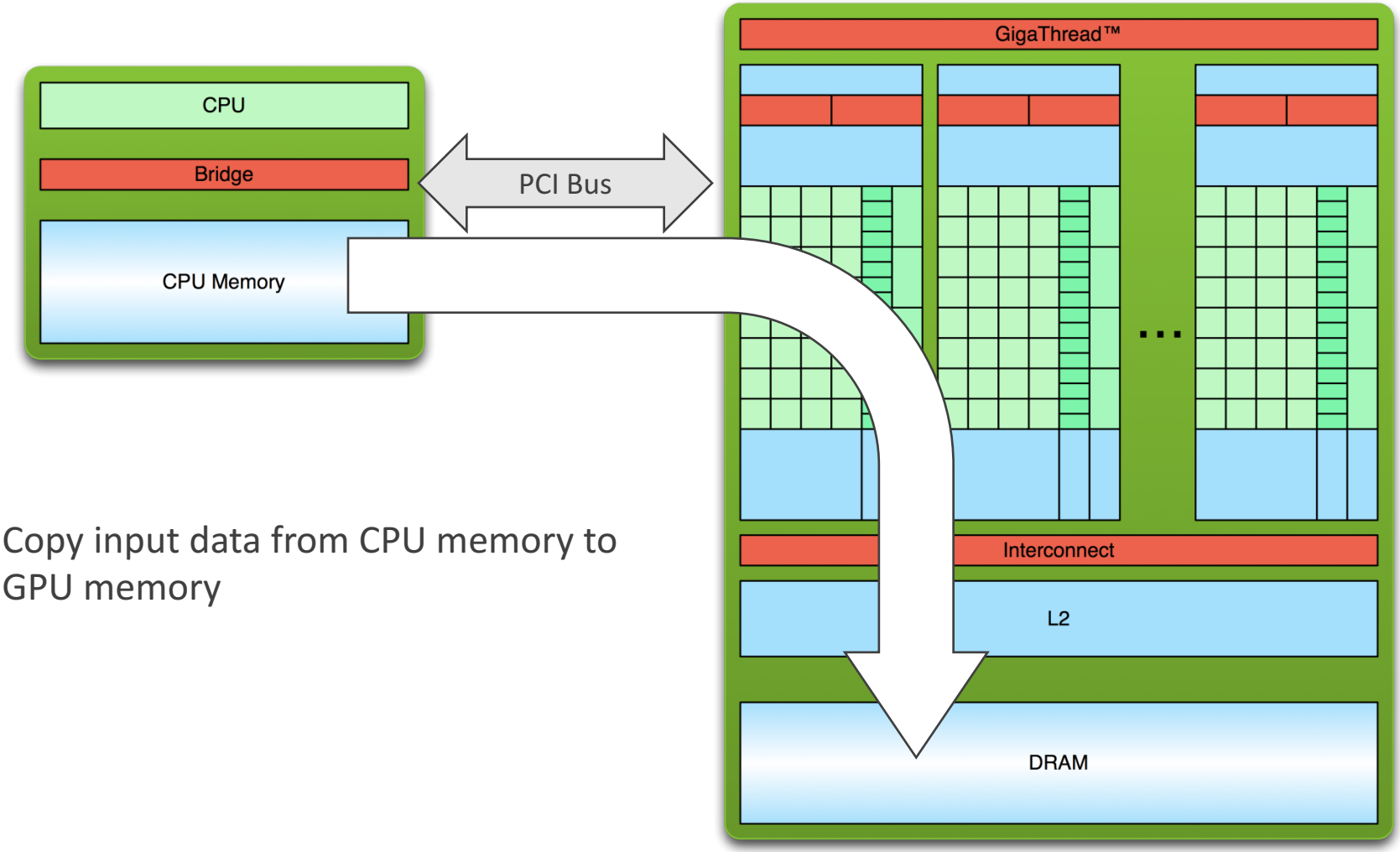
CUDA Toolkit

- Available at https://developer.nvidia.com/cuda-downloads
- Contains of
  - CUDA Driver
  - CUDA API (compiler and libraries)
  - Sample Codes

CUDA C API

- Two types of API
  - CUDA Runtime API: High level API
  - CUDA Driver API: Low level API
- CUDA C code is essentially C/C++ code with additional extensions.
- CUDA filename extension: .cu and .cuh
- CUDA Compiler, NVCC
  - Compatible with GCC Compiler and Intel Compiler
  - NVCC will separate CPU portion in CUDA code and send it to C/C++ Compiler for compilation. While GPU portion will be compiled by NVCC itself. Then, NVCC will combine CPU and GPU object files into an executable binary.
  - `$ nvcc simplecuda.cu –o simplecuda`

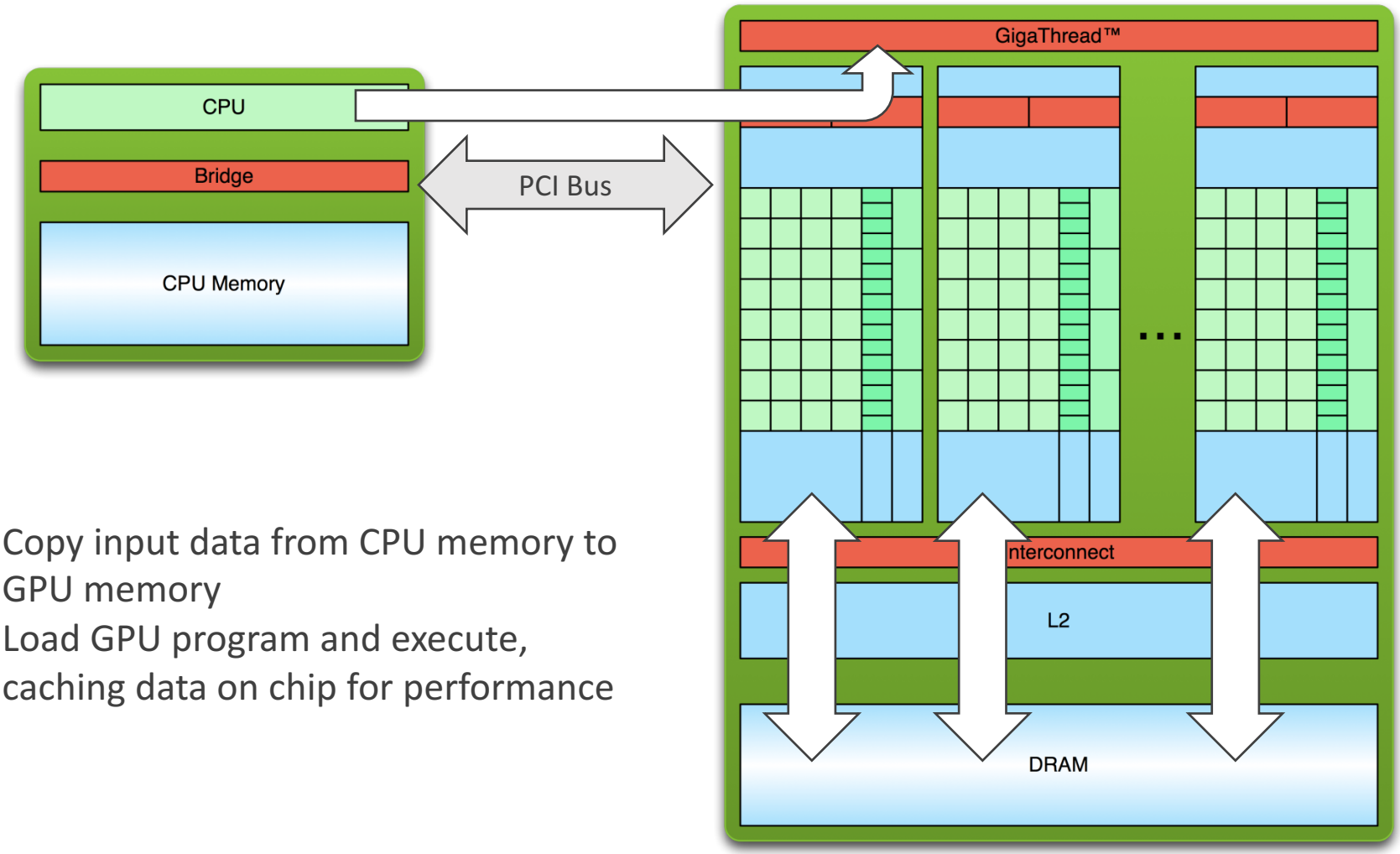
Typical CUDA Program Flow



1. Copy input data from CPU memory to GPU memory

Contents credit to NVIDIA

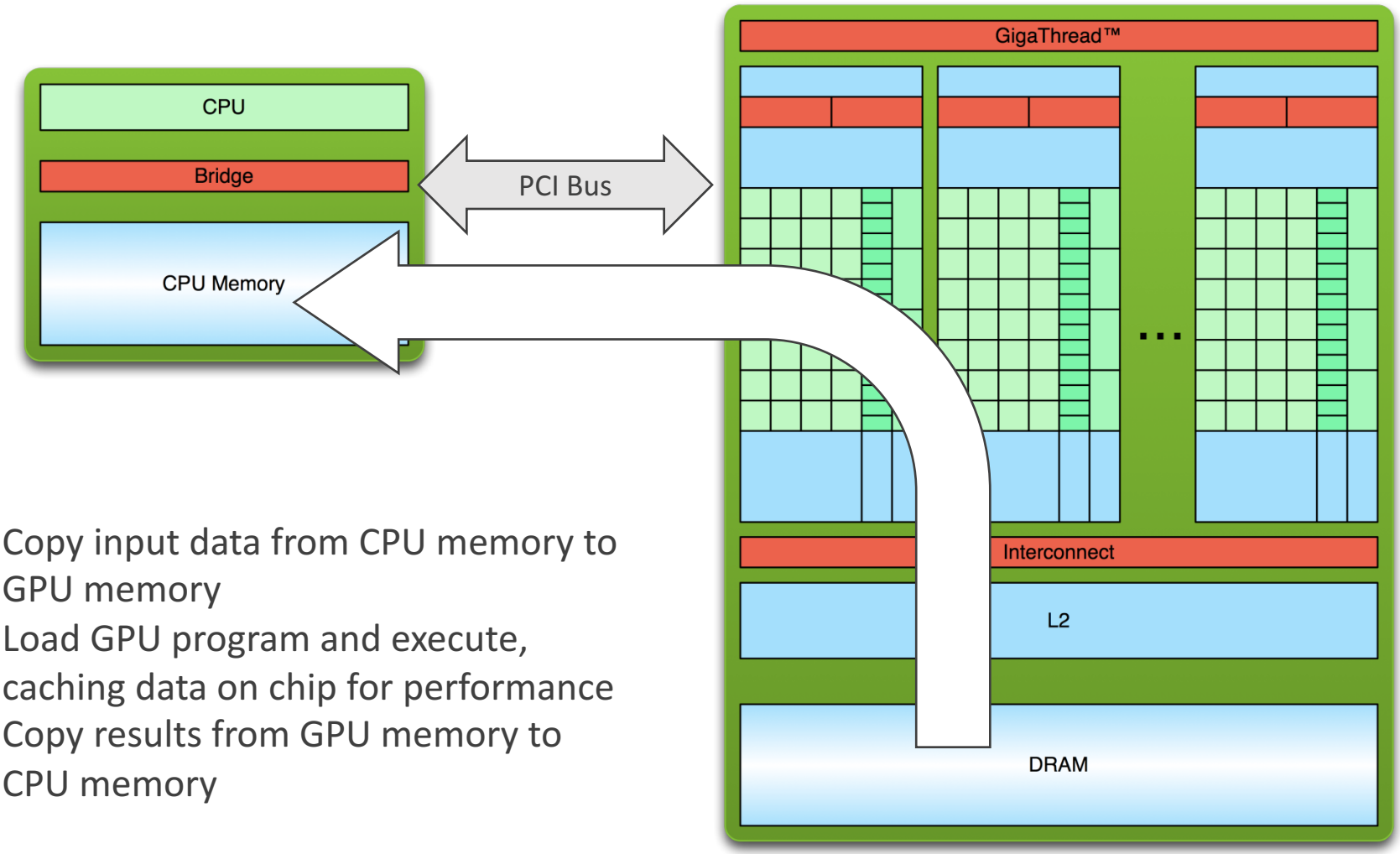
Typical CUDA Program Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Contents credit to NVIDIA

Typical CUDA Program Flow

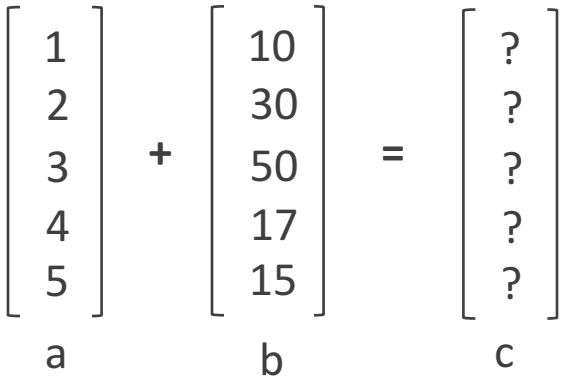


1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Contents credit to NVIDIA

Simple CUDA Code

- Vector Addition

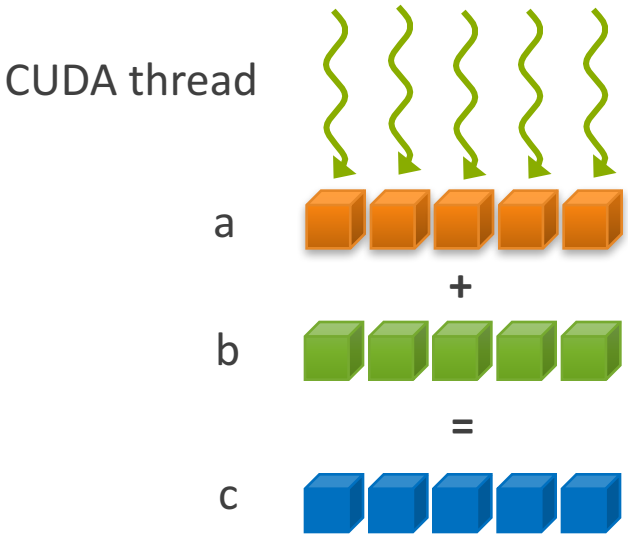


- C++ Implementation

```
for( int i = 0; i < N; i++ )
{
        c[i] = a[i] + b[i];
}
```

Simple CUDA Code

- CUDA Implementation



Simple CUDA Code

```
#include <stdio.h>

__global__ void vector_add(int *a, int *b, int *c)
{
        /* insert code to calculate the index properly
        using blockIdx.x, blockDim.x, threadIdx.x */
        int globalId = blockIdx.x * blockDim.x + threadIdx.x;
        c[globalId] = a[globalId] + b[globalId];
}

/* experiment with N */
/* how large can it be? */
#define N (512)
#define THREADS_PER_BLOCK 512

int main()
{
        int *a, *b, *c;
        int *d_a, *d_b, *d_c;
        int size = N * sizeof( int );
        int NBLOCK = (N + (THREADS_PER_BLOCK-1)) /
                        THREADS_PER_BLOCK;
```

Kernel function

Main function

Pointer vars

Simple CUDA Code

```
/* allocate space for device copies of a, b, c */
cudaMalloc( (void **) &d_a, size );
cudaMalloc( (void **) &d_b, size );
cudaMalloc( (void **) &d_c, size );

/* allocate space for host copies of a, b, c and
setup input values */
a = (int *)malloc( size );
b = (int *)malloc( size );
c = (int *)malloc( size );

for( int i = 0; i < N; i++ )
{
        a[i] = b[i] = i;
        c[i] = 0;
}
/* copy inputs to device */
/* fix the parameters needed to copy data to
the device */
cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice );
```

Allocate GPU memory

Allocate CPU memory

Copy data from CPU mem to GPU mem

Simple CUDA Code

```
        /* launch the kernel on the GPU */
        /* insert the launch parameters to launch
        the kernel properly using blocks and threads */
        vector_add<<< NBLOCK, THREADS_PER_BLOCK >>>(
        d_a, d_b, d_c );

        /* copy result back to host */
        /* fix the parameters needed to copy data back to
        the host */
        cudaMemcpy( c, d_c, size, cudaMemcpyDeviceToHost );

        printf( "c[0] = %d\n",c[0] );
        printf( "c[%d] = %d\n",N-1, c[N-1] );
        /* clean up */
        free(a);
        free(b);
        free(c);
        cudaFree( d_a );
        cudaFree( d_b );
        cudaFree( d_c );

        return 0;
} /* end main */
```

Call kernel function

Copy data from GPU to CPU mem

Free CPU mem

Free GPU mem

- CUDA Platform
- **GPU Memory**
- CUDA Kernel
- CUDA Thread
- Device Management
- Compiling & Debugging

CUDA Memory

- On-chip memory
  - Register
  - L1 Cache
  - Texture cache
  - Constant cache
  - Shared Memory

- DRAM
  - Local Memory (cached on L1 and/or L2; depends on card capability)
  - Constant Memory (cached on constant cache)
  - Texture Memory (cached on texture cache)
  - Global Memory (cached on L2)

Global Array Variables

- Allocate space in GPU global memory

```
int *d_a, *d_b, *d_c;
int size = N * sizeof( int );
/* allocate space for device copies of a, b, c */
cudaMalloc( (void **) &d_a, size );
cudaMalloc( (void **) &d_b, size );
cudaMalloc( (void **) &d_c, size );
```

- Copy data from CPU memory to GPU memory

cudaMemcpy(dest, source, size, direction)

```
cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice );
cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice );
...
cudaMemcpy( c, d_c, size, cudaMemcpyDeviceToHost );
```

- Clear GPU memory space

```
cudaFree( d_a );
cudaFree( d_b );
cudaFree( d_c );
```

Local Variables

```
__global__ void vector_add(int *a, int *b, int *c)
{
        int globalId = blockIdx.x * blockDim.x + threadIdx.x;
        c[globalId] = a[globalId] + b[globalId];
}
```

- *blockIdx, blockDim, threadIdx* are built-in local variables.
- *index* is a declared local variable.

- All declared local variables will initially be stored in registers.
- If there are not enough registers, some variables will be pushed to local memory.
- Note: Local memory is cached in L1. Hence, the different may not be significant.

Local Array Variables in Shared Memory

- Local variable that is declared using **__shared__** will be placed in shared memory.
- Usually used for array variables.
- Useful for data sharing between threads in a block.
- In early GPU architecture (where global memory has no cache), it helps to improve performance significantly.

```
__global__ void vector_add(int *a, int *b, int *c)
{
        __shared__ int c_shared[size];

       int globalId = blockIdx.x * blockDim.x + threadIdx.x;
       c_shared[globalId] = a[globalId] + b[globalId];
       c_shared[globalId] = c_shared[globalId] * b[globalId];

       c[globalId] = c_shared[globalId];
}
```

Constant Variables

- Declared using **__constant__**
- Cannot be assigned from kernel
- Supposed to be used for read-only variables called from kernel
- Will be stored in constant memory

```
__constant__ int dd = 100;

__global__ void vector_add(int *a, int *b, int *c)
{
        __shared__ int c_shared[size];

       int globalId = blockIdx.x * blockDim.x + threadIdx.x;
       c_shared[globalId] = a[globalId] + b[globalId];

       c_shared[globalId] = c_shared[globalId] * dd;

       c[globalId] = c_shared[globalId];
}
```

- CUDA Platform
- GPU Memory
- **CUDA Kernel**
- CUDA Thread
- Device Management
- Compiling & Debugging

CUDA Kernel

- Functions in the code that will be sent to GPU and executed in parallel
- The function is declared using **__global__**

```
__global__ void vector_add(int *a, int *b, int *c)
{
        ...
}
```

- Called from host code and specified number of threads with syntax **<<<…… >>>**

```
vector_add<<< NBLOCK, THREADS_PER_BLOCK >>>( d_a, d_b, d_c );
```

- May declare multiple kernel functions.
- But invoking a kernel is expensive.

Device Function

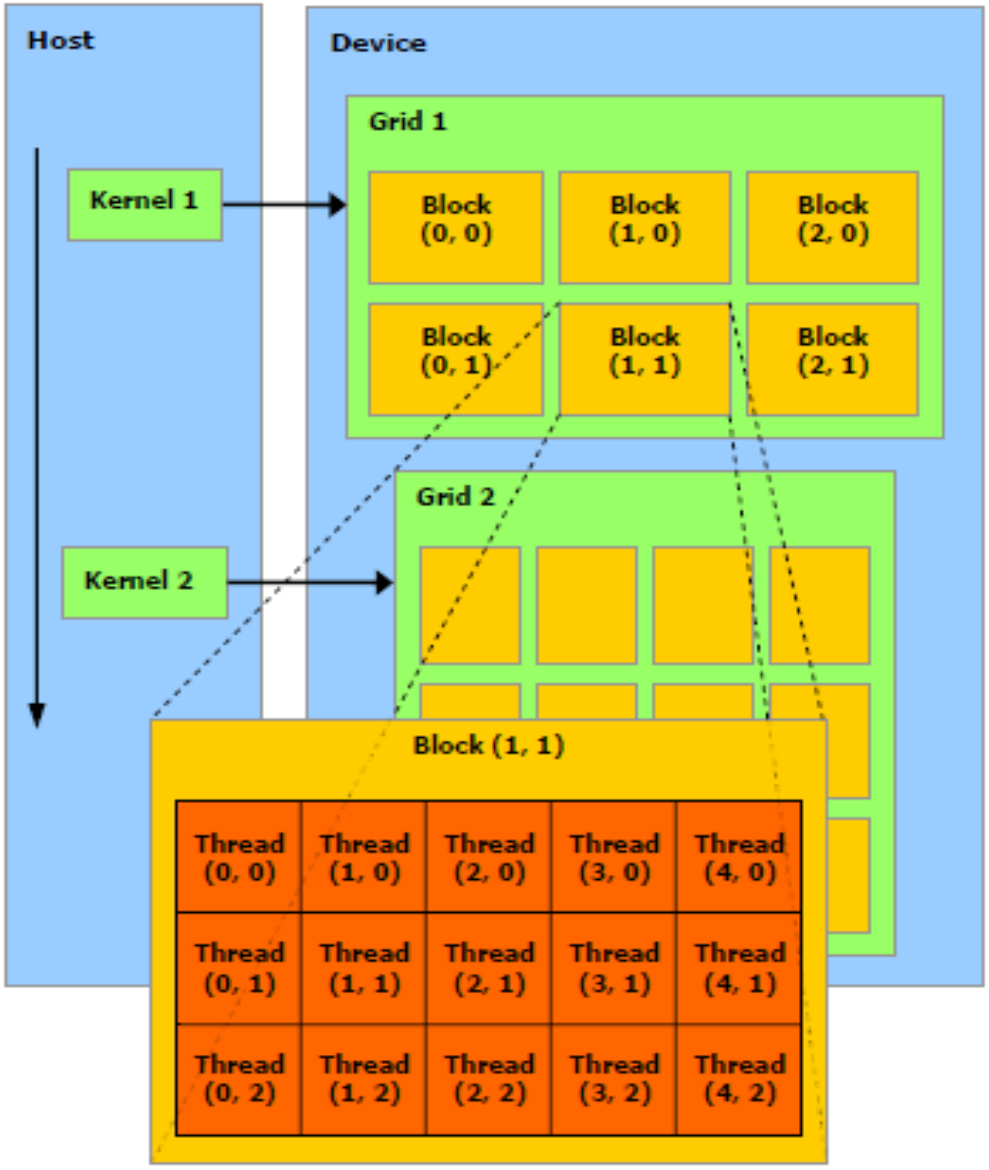
- Separate function that will be executed in GPU but called inside kernel code only.
- The function is declared using **__device__**

```
__constant__ int dd = 2;

__device__ int add_op(int ax, int bx)
{
        return (ax + bx) * dd;
}

__global__ void vector_add(int *a, int *b, int *c)
{
        int globalId = blockIdx.x * blockDim.x + threadIdx.x;
        c[globalId] = add_op( a[globalId] , b[globalId] );
}
```

- CUDA Platform
- GPU Memory
- CUDA Kernel
- **CUDA Thread**
- Device Management
- Compiling & Debugging

CUDA Thread

- Thread is managed in block and grid
- Block is a group of threads, can be in 1D, 2D, or 3D
- Grid is a group of blocks, can be in 1D, 2D or 3D



CUDA Thread

- Thread configuration is defined when executing the kernel function.

```
vector_add<<< BLOCKS_PER_GRID, THREADS_PER_BLOCK >>>( ... );
```

```
#define N (512)
#define THREADS_PER_BLOCK 512

int main()
{
        ...
        int size = N * sizeof( int );
        int NBLOCK = (N + (THREADS_PER_BLOCK-1)) / THREADS_PER_BLOCK;
        ...
        vector_add<<< NBLOCK, THREADS_PER_BLOCK >>>( d_a, d_b, d_c );
        ...
}
```

CUDA Thread

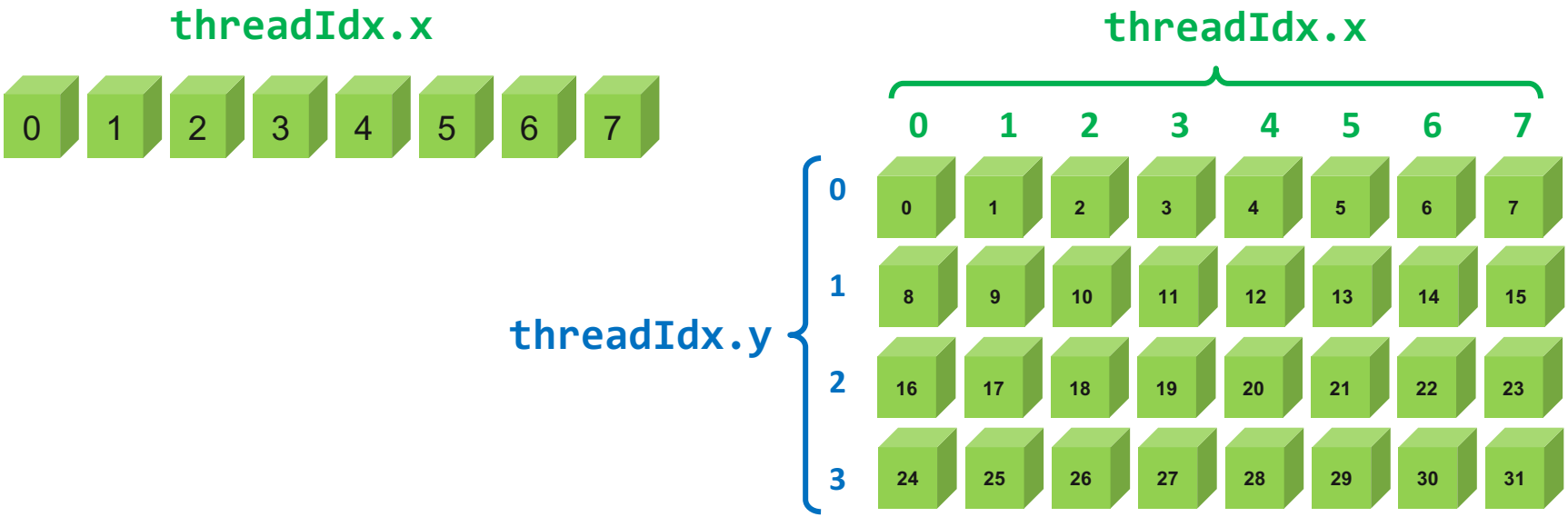
- Built-In Variables:
  - **threadIdx** : thread index in a block (in x-y-z components)
  - **blockDim** : block dimension in x-y-z component
  - **blockIdx** : block index in a grid (in x-y-z components)
  - **gridDim** : grid dimension in x-y-z component

- During execution, threads within a block are configured as groups of 32-threads, called *warp*.

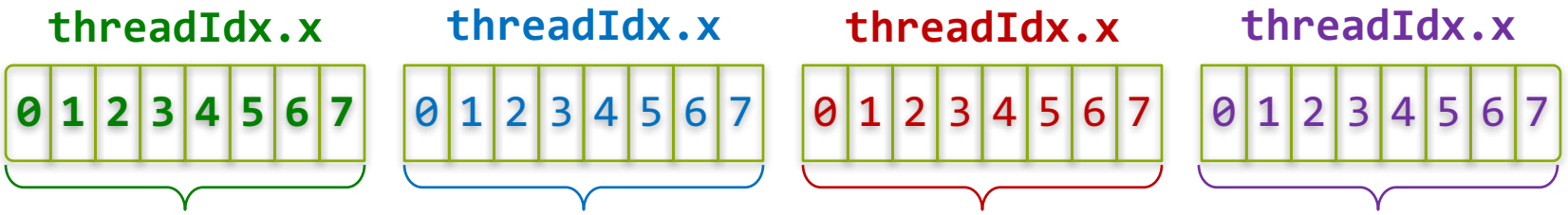
- Hence, it is recommended to have block size in multiple of 32.

CUDA Thread – Local ID

- For a 1D block, thread local ID is its x-index (**threadIdx.x**).
- For a 2D block of size (Dx, Dy), the thread local ID of a thread of index (x, y) is (**threadIdx.x** + **threadIdx.y** * **blockDim.x**);
- For a 3D block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (**threadIdx.x** + **threadIdx.y** * **blockDim.x** + **threadIdx.z** * **blockDim.x** * **blockDim.y**).



CUDA Thread – Global ID

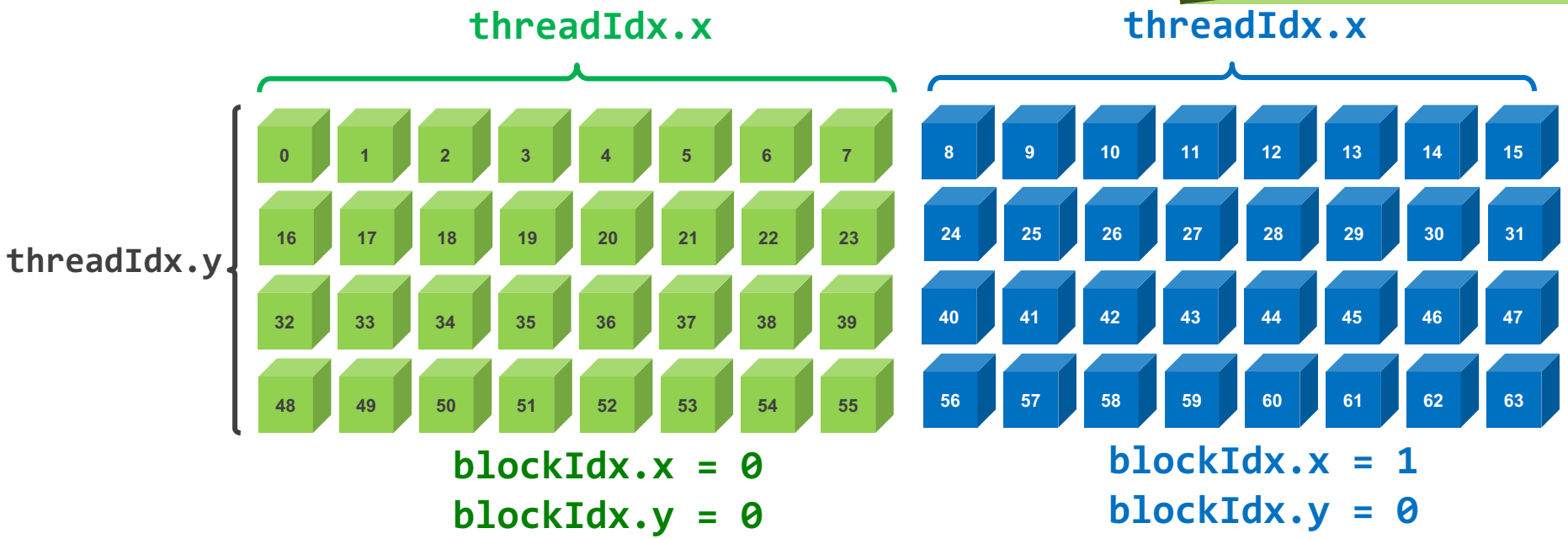


blockIdx.x = 0 **blockIdx.x = 1** **blockIdx.x = 2** **blockIdx.x = 3**

blockDim.x = 8

```
int globalId = threadIdx.x + blockIdx.x * blockDim.x;
```

CUDA Thread – Global ID




```
blockDim.x = 8          gridDim.x = 2
blockDim.y = 4          gridDim.y = 1
```

```
int global_x = threadIdx.x + blockIdx.x * blockDim.x;
int global_y = threadIdx.y + blockIdx.y * blockDim.y;

int globalId = global_x + global_y * blockDim.x * gridDim.x ;
```

Sharing Data Between Threads

- Utilized shared variables which resides in shared memory.
- Shared variable is declared using **__shared__**.
- Works in per-block basis; threads within a same block.
- Data are not visible to threads in other block.

Thread Synchronization

- **__syncthreads();** is used to coordinate communication between the threads of the same block.
- It is usually needed when we implement data sharing within threads in a block.
- To prevent RAW / WAR / WAW hazards.

```
__global__ void vector_add(int *a, int *b, int *c)
{
          __shared__ int c_shared[size];

        int globalId = blockIdx.x * blockDim.x + threadIdx.x;
        c_shared[globalId] = a[globalId] + b[globalId];
        c_shared[globalId] = c_shared[globalId] * b[globalId];

        __syncthreads();

        c[globalId] = c_shared[globalId] + c_shared[0];
}
```

- CUDA Platform
- GPU Memory
- CUDA Kernel
- CUDA Thread
- **Device Management**
- Compiling & Debugging

Device Management

- By default, CUDA code will use the first GPU (device 0)
- To select a specific device

```
cudaSetDevice(int device)
```

- To query GPUs

```
cudaGetDeviceCount(int *count)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);

cudaSetDevice(deviceCount-1);

int device;
cudaDeviceProp prop;

cudaGetDevice(&device);
cudaGetDeviceProperties(&prop, device);
```

Coordinating Host and Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

`cudaMemcpy()`: Blocks the CPU until the copy is complete. Copy begins when all preceding CUDA calls have completed.

`cudaDeviceSynchronize()`: Blocks the CPU until all preceding CUDA calls have completed.

CUDA Platform

GPU Memory

CUDA Kernel

CUDA Thread

Device Management

Compiling & Debugging

Compiling

- If using single cuda file

```
nvcc <cu_file> -o <executable_binary>
E.g: nvcc simplecuda.cu –o simplecuda
```

- If the main function and kernel function are in separate separate files:

```
nvcc –c <cu_file>
g++ <cppfile> <cu_object_file> -o <exe_binary>

E.g:
nvcc –c vectoradd.cu
g++ main.cpp vectoradd.o -o simplecuda
```

Getting Error

- All CUDA API calls return an error code (`cudaError_t`)

- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void);
```

- Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t);

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

IDE, Debugging, and Profiling

- IDE: *Nsight* – eclipse based IDE with CUDA debugger and GUI profiling tool included.
- Debugging tool: *CUDA-GDB* – works like GNU Debugger.
- Profiling tool: *nvprof* – a command line profiling tool.

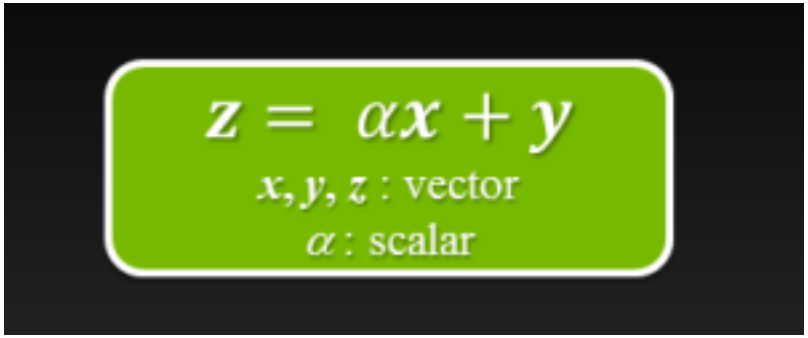
Advanced Features (Not Covered)

- Storing data as 2D arrays
- Optimization
  - Shared Memory
  - Asynchronous Functions
  - Memory Coalescing
  - Single/Double Precision Functions
- Unified Memory
- Dynamic Parallelization
- Atomic Operation
- Libraries

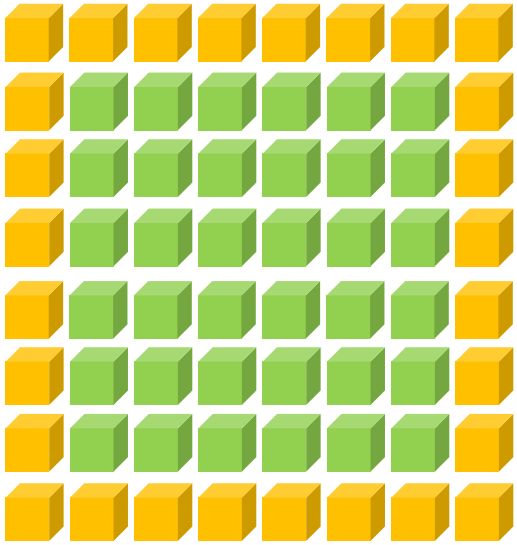
- Sources for self-study:
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide
  - http://docs.nvidia.com/cuda/cuda-c-best-practices-guide
  - Developer forum, https://devtalk.nvidia.com

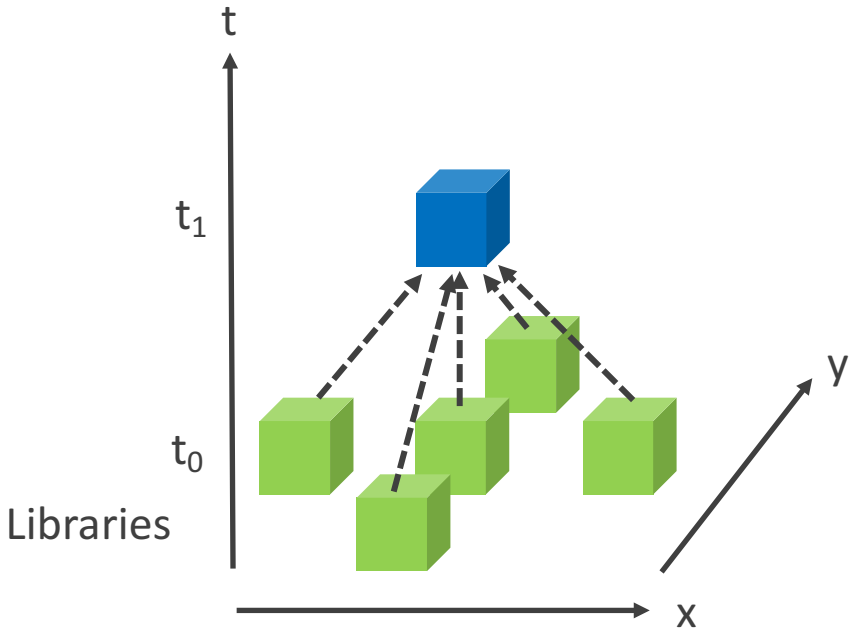
Exercise #1

- SAXPY (Single-Precision AX Plus Y)



Exercise #2





Boundary Condition

$y_{(i,j)} = x_{(i,j)}$

$y_{(i,j)} = x_{(i,j)} + x_{(i+1,j)} + x_{(i-1,j)} + x_{(i,j+1)} + x_{(i,j-1)}$

Exercise #2

Input:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Expected Output:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	5	5	5	5	5	5	5	5	5	5	5	5	5	5	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Looking For GPU Computing Solutions ?

support@novaglobal.com.sg

NovaGlobal Pte Ltd
Green & Scientific Computing Solutions
http://novaglobal.com.sg

- Established in 1996
- Operating in ASEAN region
  - Primarily in Singapore, Malaysia, Indonesia, Thailand and Vietnam
- Solution provider for
  - High Performance Computing
  - Accelerated GPU Computing
  - Cloud/Virtualization
- Platform-Independent
- Partnering with Technology Leaders