HPC Software Framework- CUDA and OpenACC

Wednesday, Nov 2, 2022
Session

DATAEVER CONSULTING

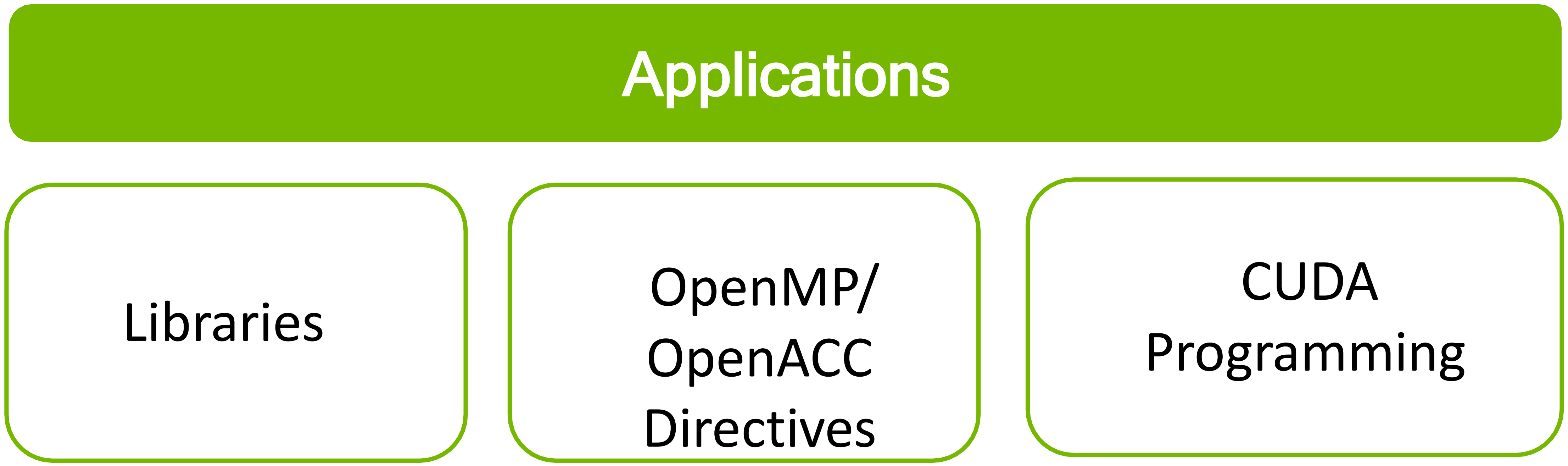
Prerequisites

- You (probably) need experience with C or C++
- You don’t need GPU experience
- You don’t need parallel programming experience
- You don’t need graphics experience

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors

3 WAYS TO PROGRAM GPU based Accelerators



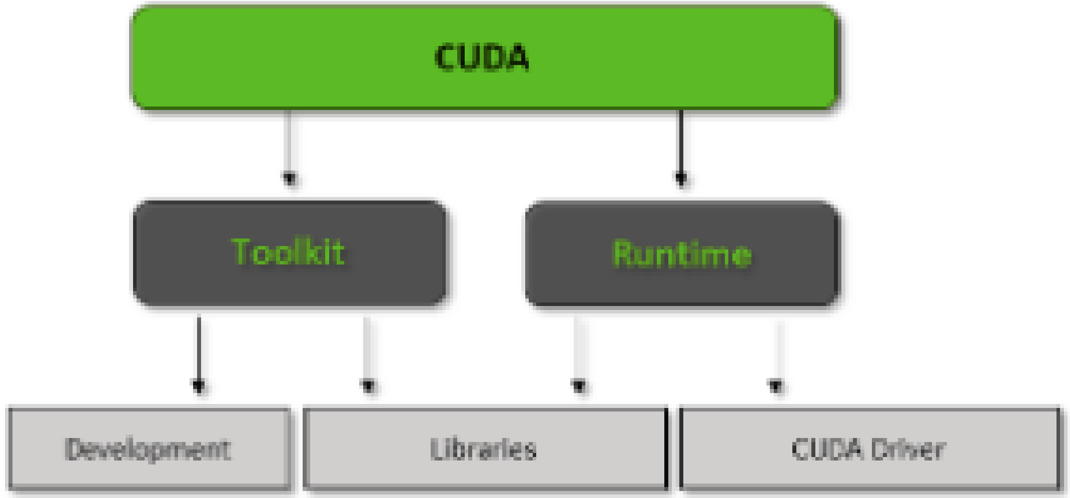
What is CUDA ?

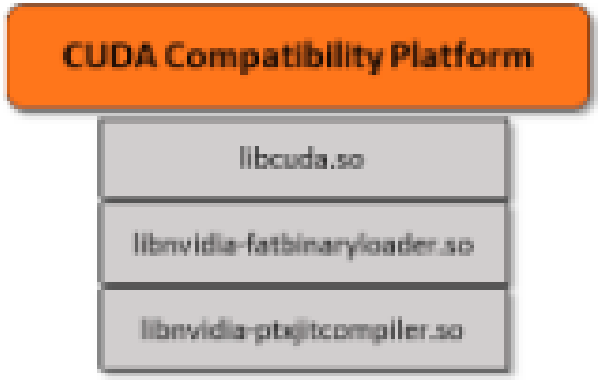
- CUDA is the name of NVIDIA’s parallel computing architecture for GPUs
- NVIDIA provides a complete toolkit for programming the CUDA architecture that includes the compiler, debugger, profiler, libraries
- The CUDA architecture supports standard languages such as C, C++ and Fortran, and APIs for GPU Computing, such as OpenCL and DirectCompute.

https://www.nvidia.com/en-us/geforce/technologies/cuda/faq/

CUDA compatibility platform

CUDA Driver and Toolkit




CUDA Compatibility Platform
libcuda.so
libnvidia-fatbinaryloader.so
libnvidia-ptxjitcompiler.so

CUDA Compute Capability

The *compute capability(CC)* describes the features of the h/w

It reflects the set of instructions supported by the device
also

+ the maximum number of threads per block
+ the number of registers per multiprocessor

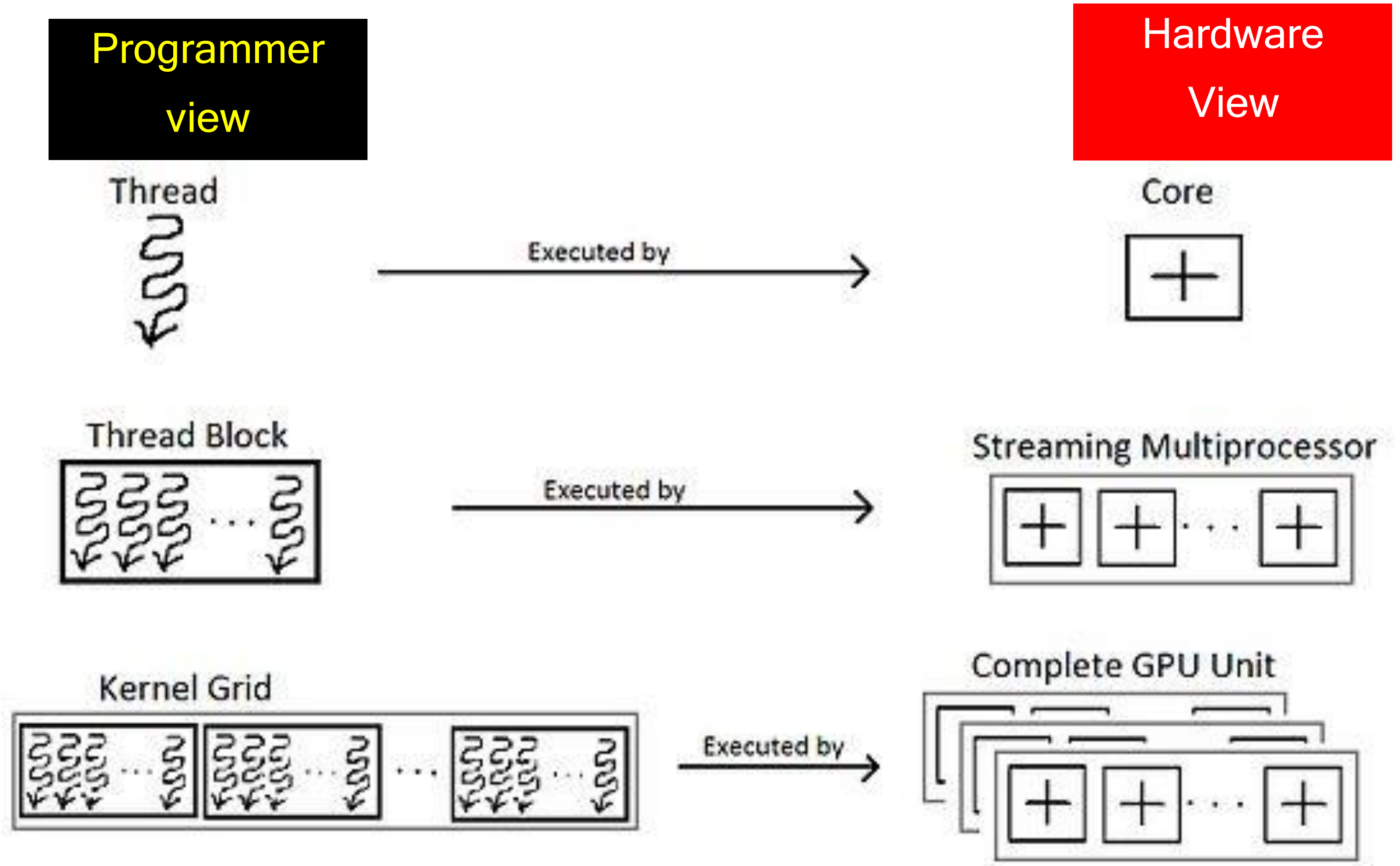
Higher compute capability versions are supersets of lower versions, so they are backward compatible.

The *CC* of the GPU in the device can be queried programmatically using `deviceQuery`

GPU configuration - Lab Environment

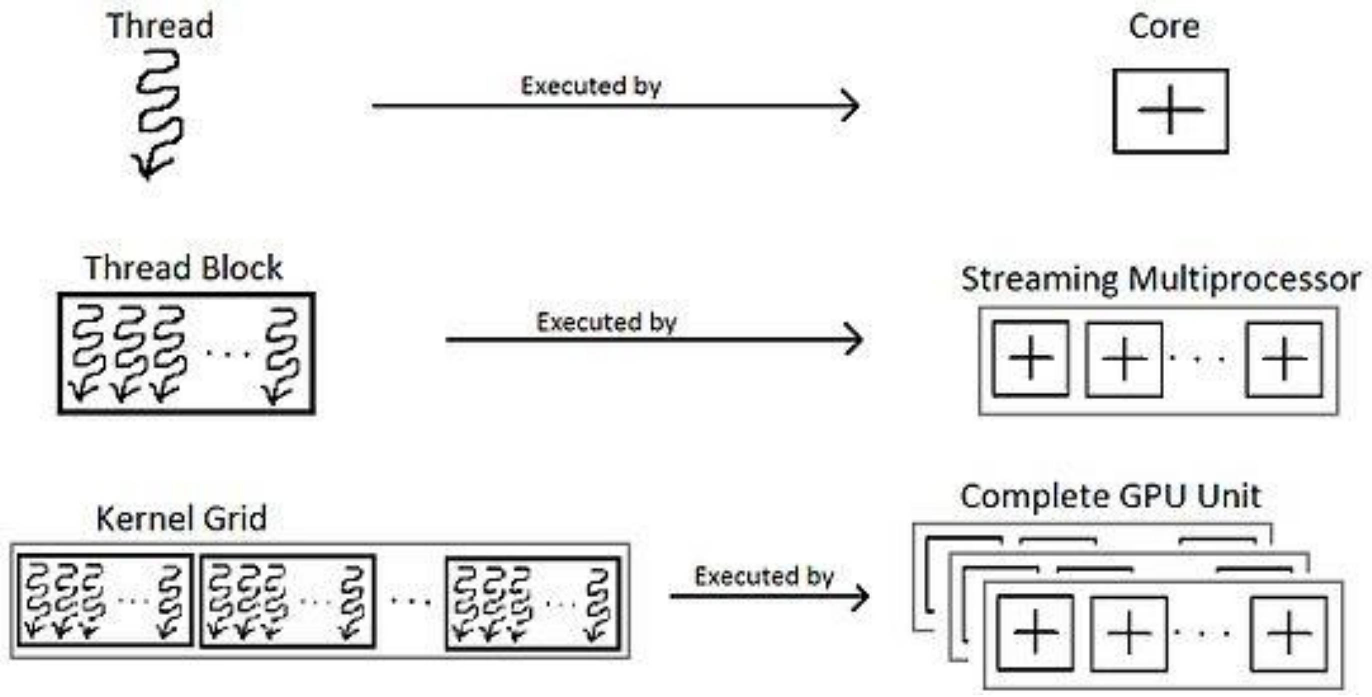
```
Device 0: "GeForce 940MX“, , NumDevs = 1
  CUDA Driver Version / Runtime Version          11.0 / 11.0
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2004 MBytes (2101870592 bytes)
  ( 3) Multiprocessors, (128) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            1242 MHz (1.24 GHz)
  Memory Clock rate:                             1001 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

Programmer perspective vs Hardware perspective



Thread, Blocks and Warps

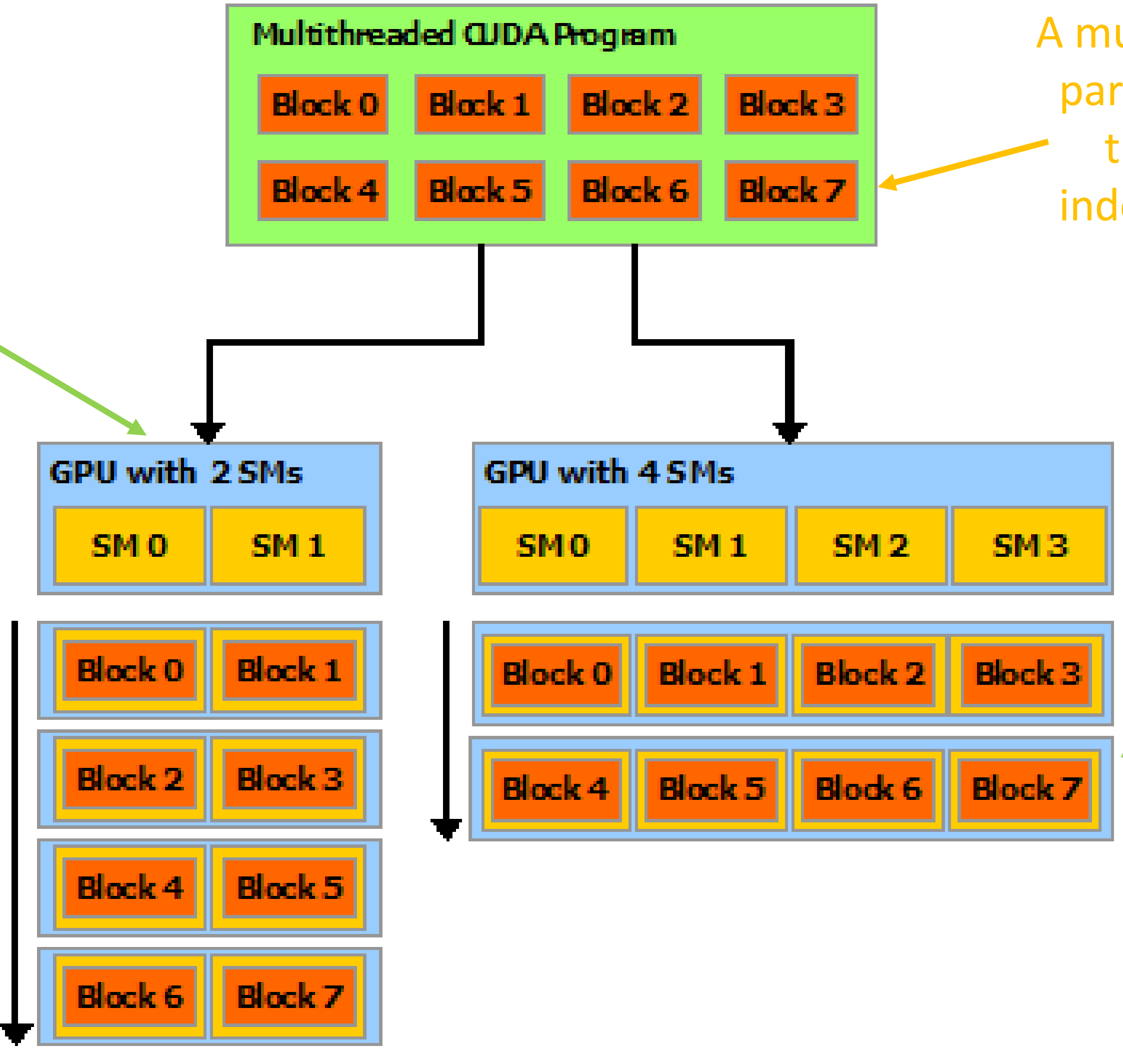
- A thread block is composed of ‘warps’. A warp is a set of 32 threads within a thread block such that all the threads in a warp execute the same instruction. These threads are selected serially by the SM
- Once a thread block is launched on a multiprocessor (SM), all of its warps are resident until their execution finishes. Thus a new block is not launched



Automatic Scalability with CUDA

A GPU is built around an array of Streaming Multiprocessors (SMs).

Multiple Thread Blocks Can be assigned to a single SM



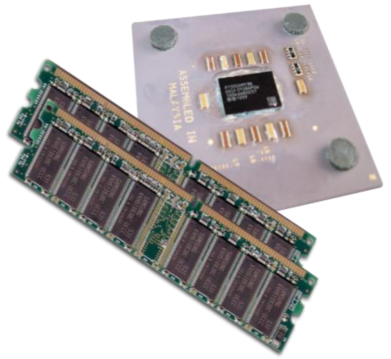
A multithreaded program is partitioned into blocks of threads that execute independently from each other

A GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors

DATAEVER CONSULTING

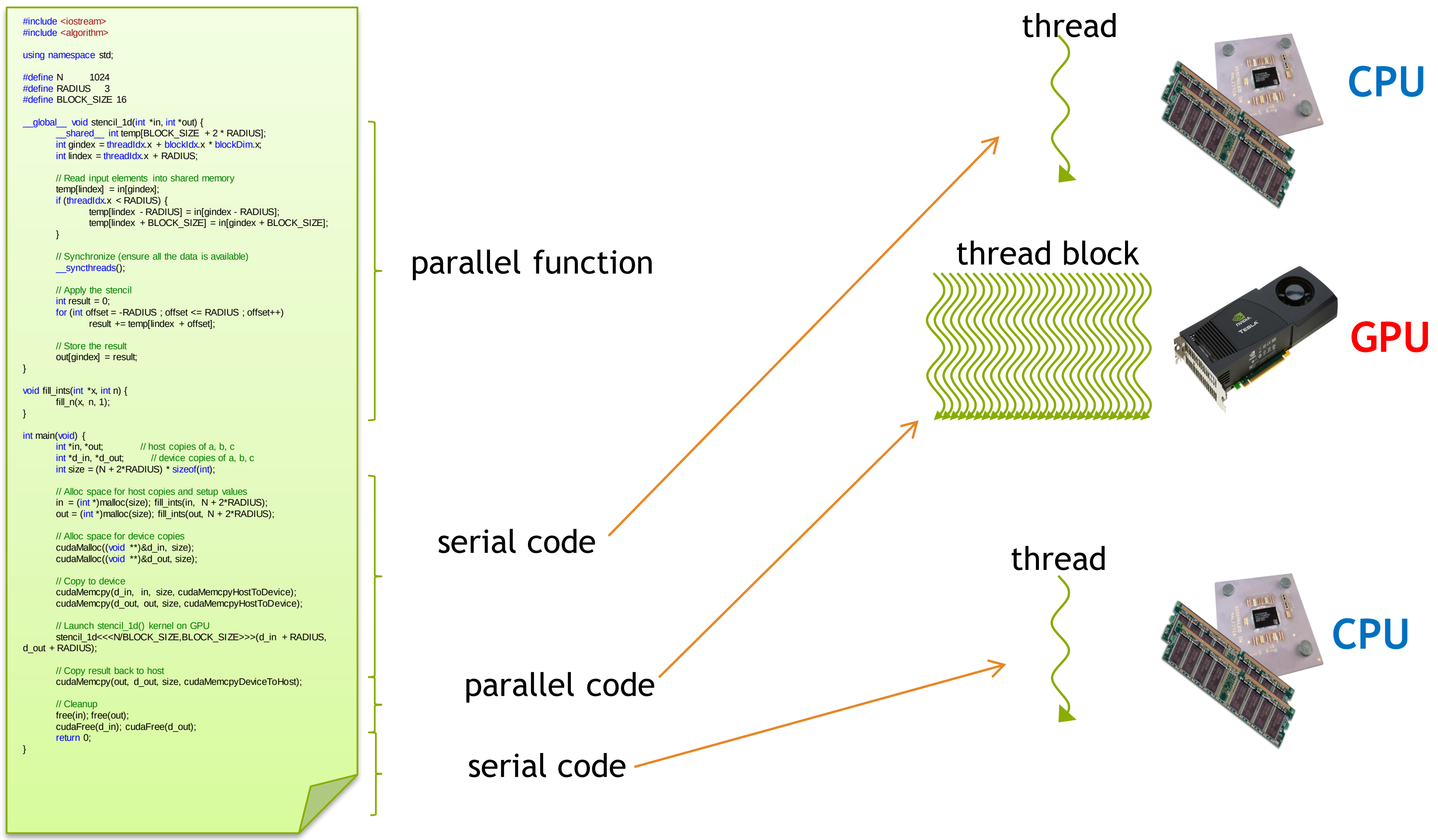
Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)


Host

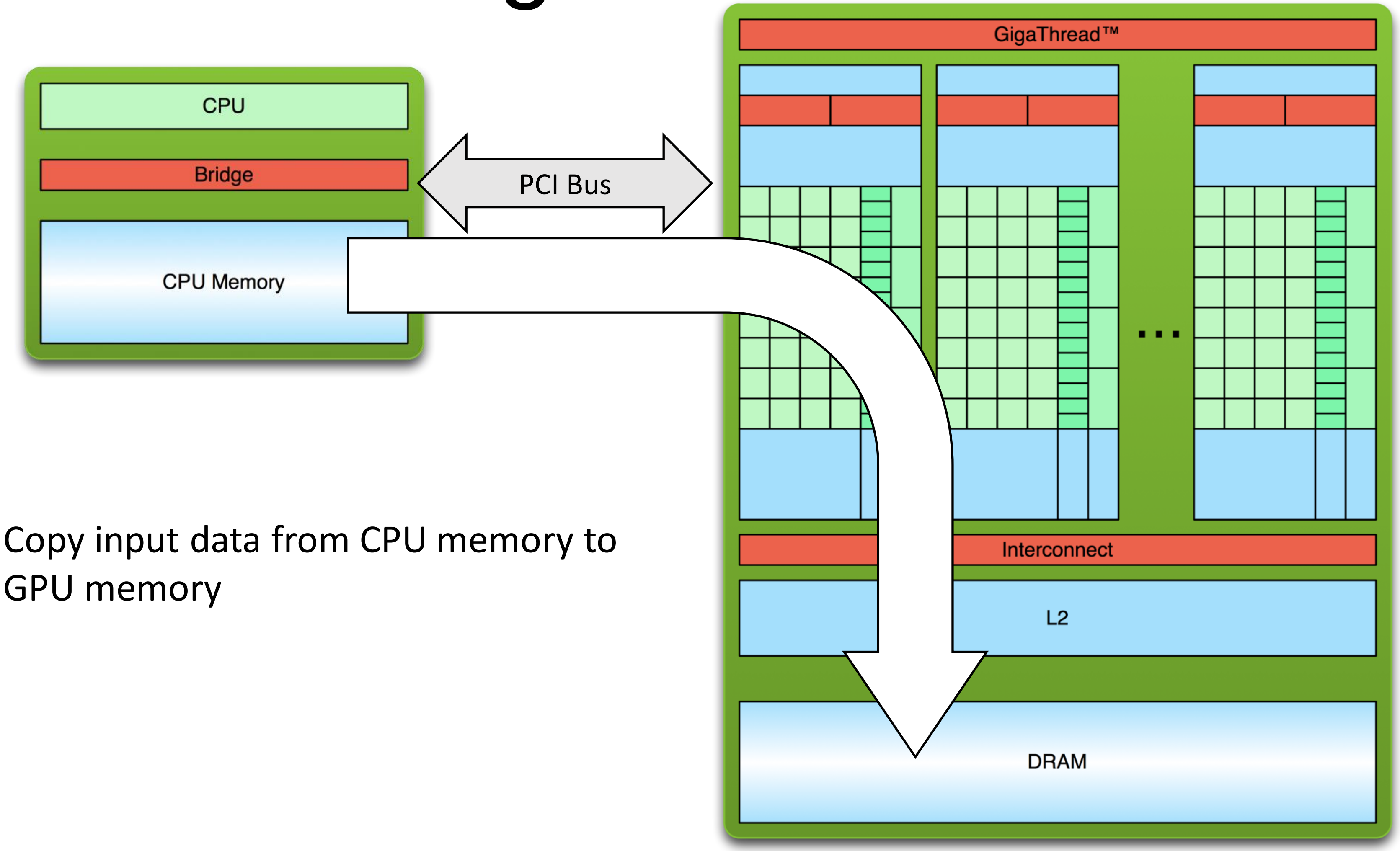
Device

Heterogeneous Computing



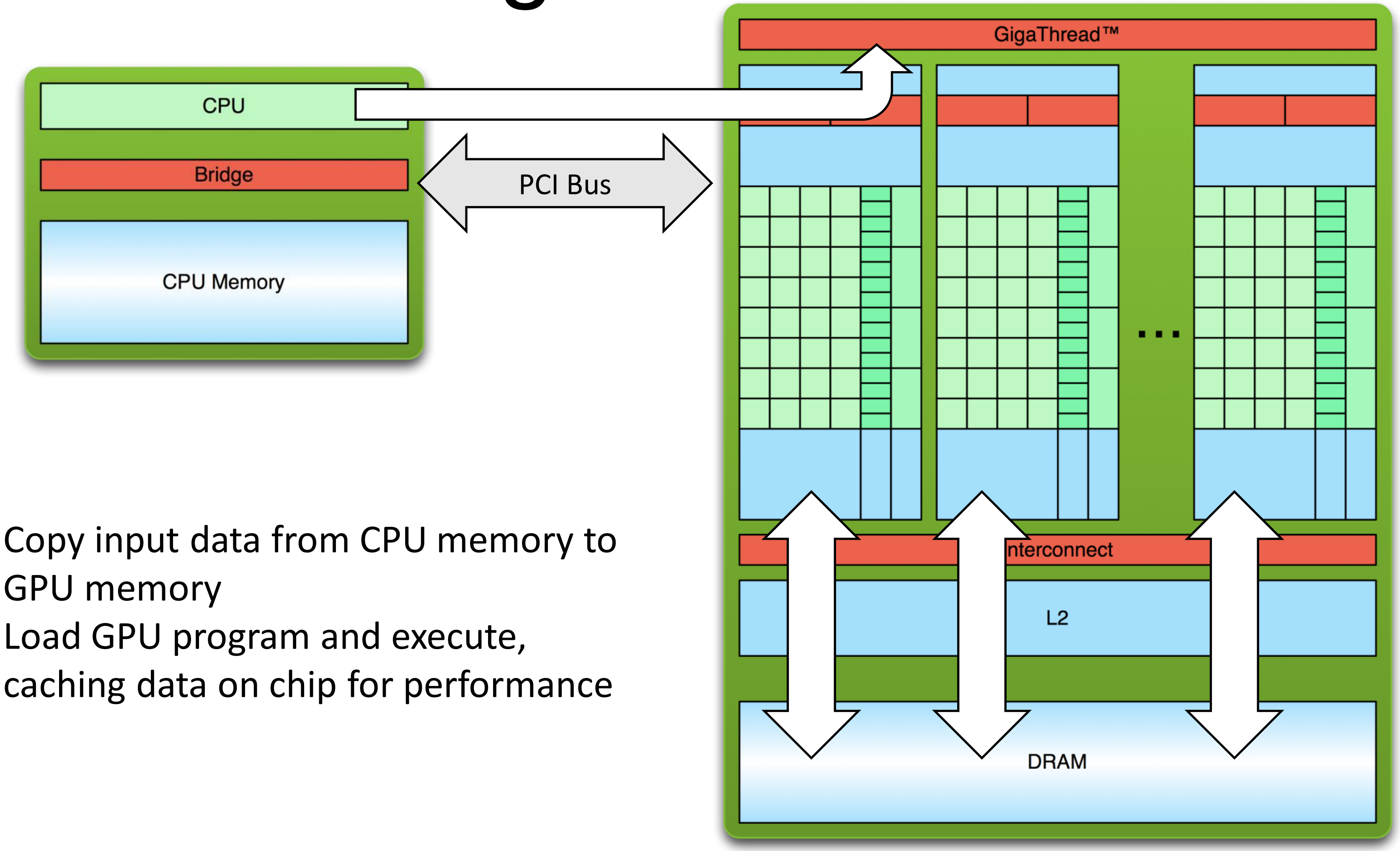
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__  void stencil_1d(int *in, int *out) {
          __shared__  int temp[BLOCK_SIZE  + 2 * RADIUS];
          int gindex = threadIdx.x + blockIdx.x * blockDim.x;
          int lindex = threadIdx.x + RADIUS;

          // Read input elements into shared memory
          temp[lindex] = in[gindex];
          if (threadIdx.x < RADIUS) {
                    temp[lindex - RADIUS] = in[gindex - RADIUS];
                    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
          }

          // Synchronize (ensure all the data is available)
          __syncthreads();

          // Apply the stencil
          int result = 0;
          for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
                    result += temp[lindex + offset];

          // Store the result
          out[gindex] = result;
}

void fill_ints(int *x, int n) {
          fill_n(x, n, 1);
}

int main(void) {
          int *in, *out;            // host copies of a, b, c
          int *d_in, *d_out;        // device copies of a, b, c
          int size = (N + 2*RADIUS) * sizeof(int);

          // Alloc space for host copies and setup values
          in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
          out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

          // Alloc space for device copies
          cudaMalloc((void **)&d_in, size);
          cudaMalloc((void **)&d_out, size);

          // Copy to device
          cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
          cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

          // Launch stencil_1d() kernel on GPU
          stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

          // Copy result back to host
          cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

          // Cleanup
          free(in); free(out);
          cudaFree(d_in); cudaFree(d_out);
          return 0;
}
```

Simple Processing Flow



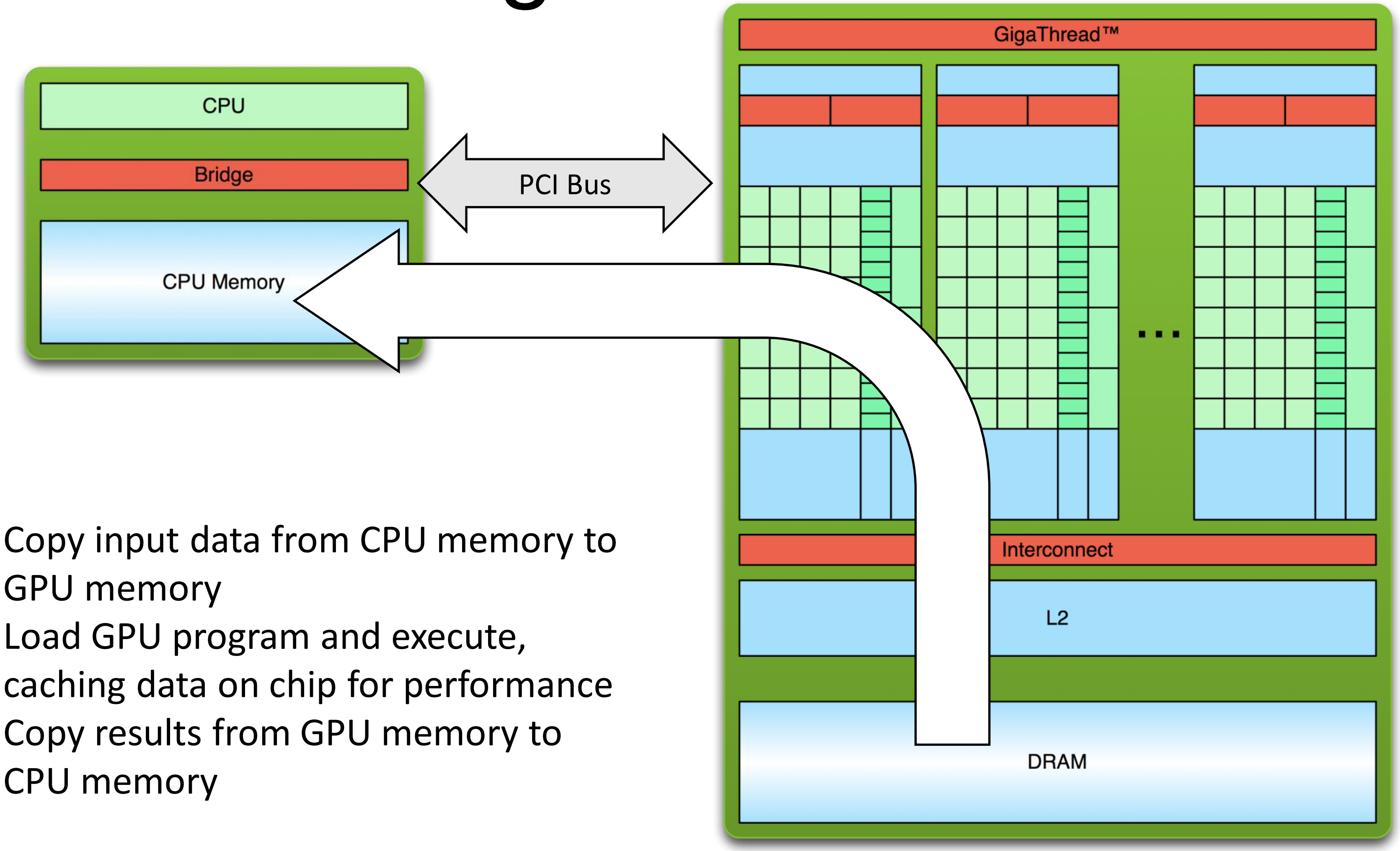
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



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

Simple Processing 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

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.
cu
$ a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

- Two new syntactic elements...

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

Hello World! with Device COde

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

> cat add.cu

```
__global__ void add(int *a, int *b, int
*c)
        {
         *c = *a + *b;
        }
#include<stdio.h>
//int add(int *a,int *b,int *c);
int main(void)
        {
int a, b, c;  // host copies of a, b, c

int *d_a, *d_b, *d_c; //device copies
of a, b, c

int size = 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);
```

> cat add.cu (continued)

```
// Setup input values
a = 2;
b = 7;

// Copy inputs to device
cudaMemcpy(d_a, &a, size,
cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size,
cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size,
cudaMemcpyDeviceToHost);
// Cleanup
cudaFree(d_a); cudaFree(d_b);
cudaFree(d_c);
printf("Sum of a and b = %d \n",c);
return 0;
        }
```

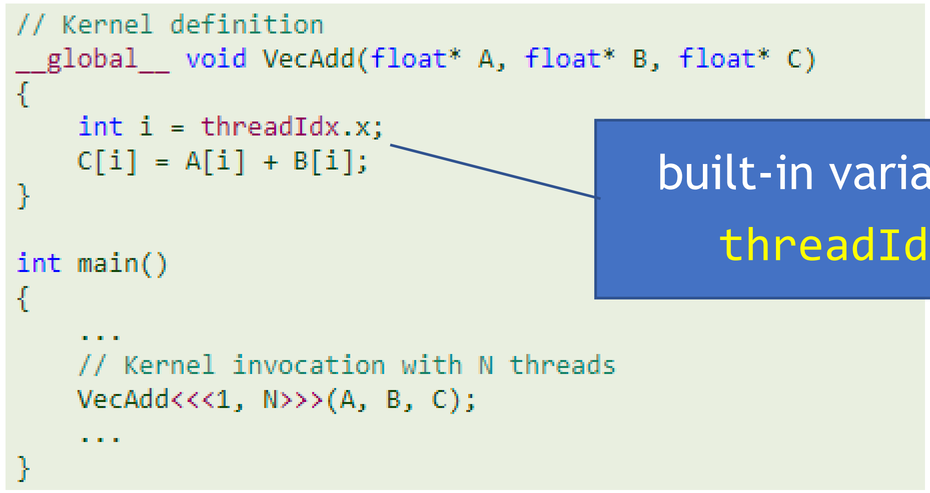
DATAEVER CONSULTING

login to dhruv

source go2EX3-ADD

source go-compile-link

CUDA Execution Model



```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

built-in variable

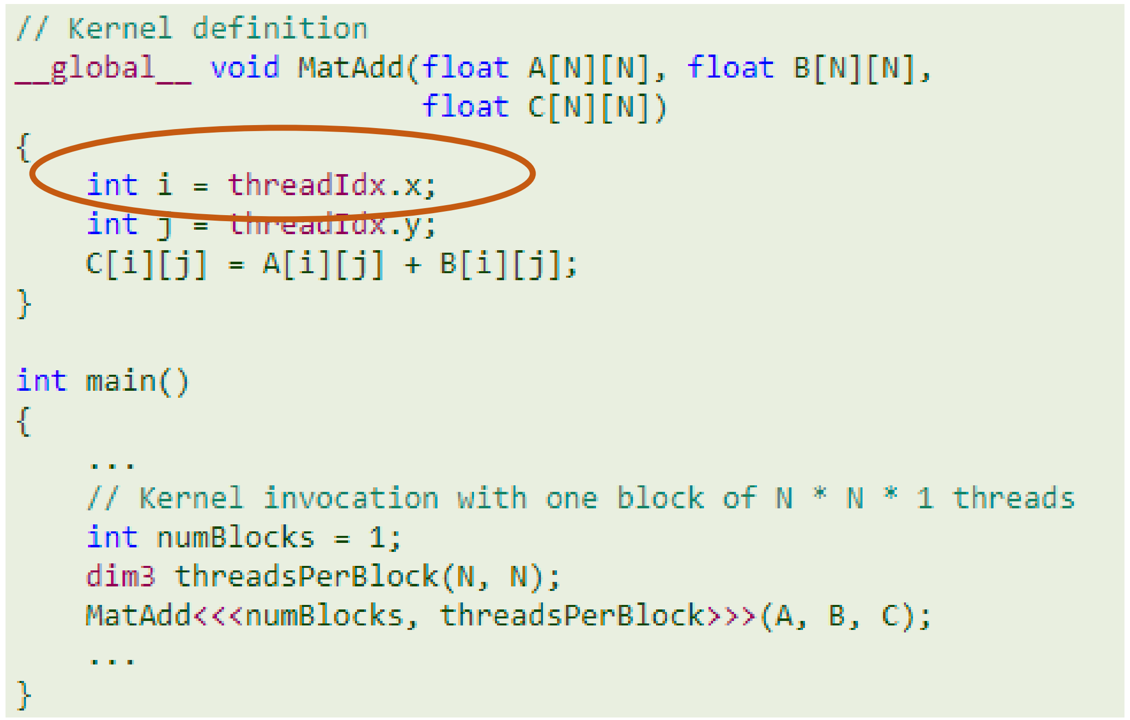
`threadIdx`

- Operational view of how instructions are executed CUDA threads are executed on a specific computing architecture

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Identifying threads using `threadIdx`

sample code



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

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- `threadIdx` is a 3-component vector
- Using this threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional block of threads, called a thread block
- This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume

EX6-PAR-THREADID.cu

```
__global__ void add(int *a,int *b,int*c)

    {c[threadIdx.x] = a[threadIdx.x] +
                      b[threadIdx.x];}

//-----------------------------------
#define N 64
#include <stdio.h>
    int main(void) {
        int *a, *b, *c;
// host copies of a, b, c
        int *d_a, *d_b, *d_c;
// device copies of a, b, c
        int size = N * sizeof(int);
        // Alloc space for device
copies of a, b, c
        cudaMalloc((void **)&d_a,
size);
        cudaMalloc((void **)&d_b,
size);
        cudaMalloc((void **)&d_c,
size);
// Alloc 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 x = 0; x < N; x++)
            {
             a[x]=x;
             b[x]=x;
            }
         // Copy inputs to device
        cudaMemcpy(d_a, a, size,
cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size,
cudaMemcpyHostToDevice);
//Launch add() kernel on GPU with N
threads per Block
        //add<<<Number of Blocks,
Number of Threads per Block>>>
        add<<<1,N>>>(d_a, d_b, d_c);
        // Copy result back to host
        cudaMemcpy(c, d_c, size,
cudaMemcpyDeviceToHost);
//Cleanup
free(a);free(b);free(c);
```

DATAEVER CONSULTING

```
login to sambath@dhruv

source go2EX6-PAR-THREADID

cat go-compile-run

source go-compile-run
```

Hello World! with Device Code

```
__global__ void mykernel(void){
}

int main(void) {
        mykernel<<<1,1>>>();
        printf("Hello World!\n");
        return 0;
}
```

Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

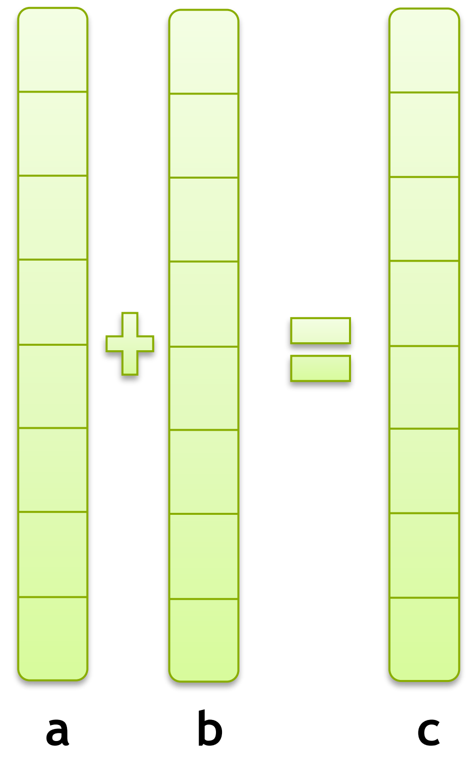
- `mykernel()` does nothing, somewhat anticlimactic!

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!

- We need a more interesting example…

- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- Let’s take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        int size = 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);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

Moving to Parallel

- GPU computing is about massive parallelism
    - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();

add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

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

- On the device, each block can execute in parallel:

Block 0

```
c[0]  = a[0] + b[0];
```

Block 1

```
c[1]  = a[1] + b[1];
```

Block 2

```
c[2]  = a[2] + b[2];
```

Block 3

```
c[3]  = a[3] + b[3];
```

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

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

- Let’s take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c                  // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

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

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

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Review (1 of 2)

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU

- Using `__global__` to declare a function as device code
  - Executes on the device
  - Called from the host

- Passing parameters from host code to a device function

Review (2 of 2)

- Basic device memory management
  - `cudaMalloc()`
  - `cudaMemcpy()`
  - `cudaFree()`

- Launching parallel kernels
  - Launch `N` copies of `add()` with `add<<<N,1>>>(…);`
  - Use `blockIdx.x` to access block index

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

INTRODUCING THREADS

CUDA Threads

- Terminology: a block can be split into parallel threads

- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

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

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`…

Vector Addition Using Threads: `main()`

```
#define N 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;              // device copies of a, b, c
    int size = N * sizeof(int);

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

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

Vector Addition Using Threads: `main()`

```
     // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

COMBINING THREADS AND BLOCKS

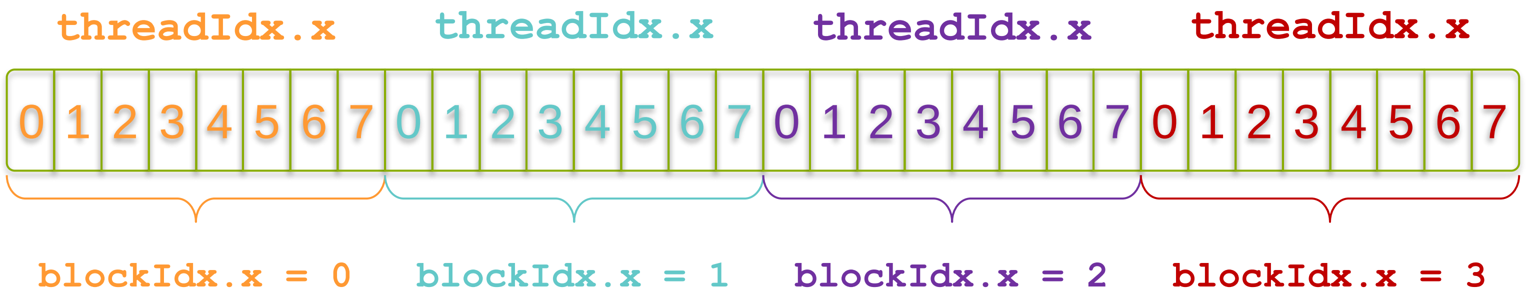
CONCEPTS

Combining Blocks and Threads

- We’ve seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let’s adapt vector addition to use both blocks and threads
- Why? We’ll come to that…
- First let’s discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
    - Consider indexing an array with one element per thread (8 threads/block)



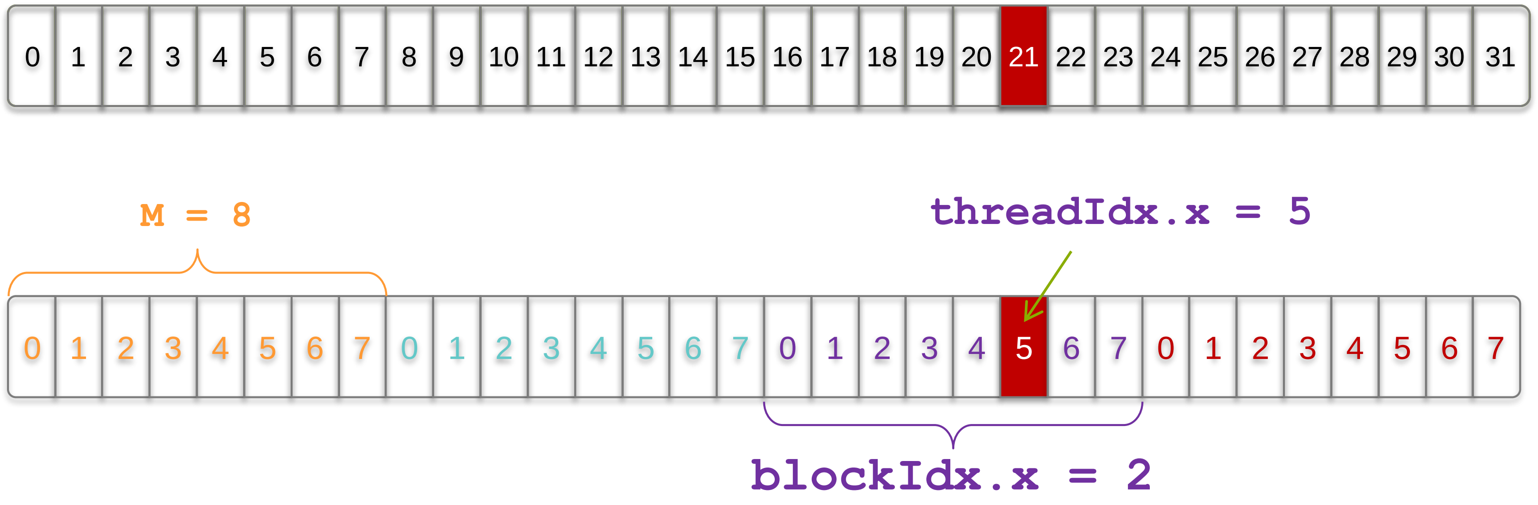
- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

DATAEVER CONSULTING

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2      * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;             // device copies of a, b, c
    int size = N * sizeof(int);

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

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

Addition with Blocks and Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

References

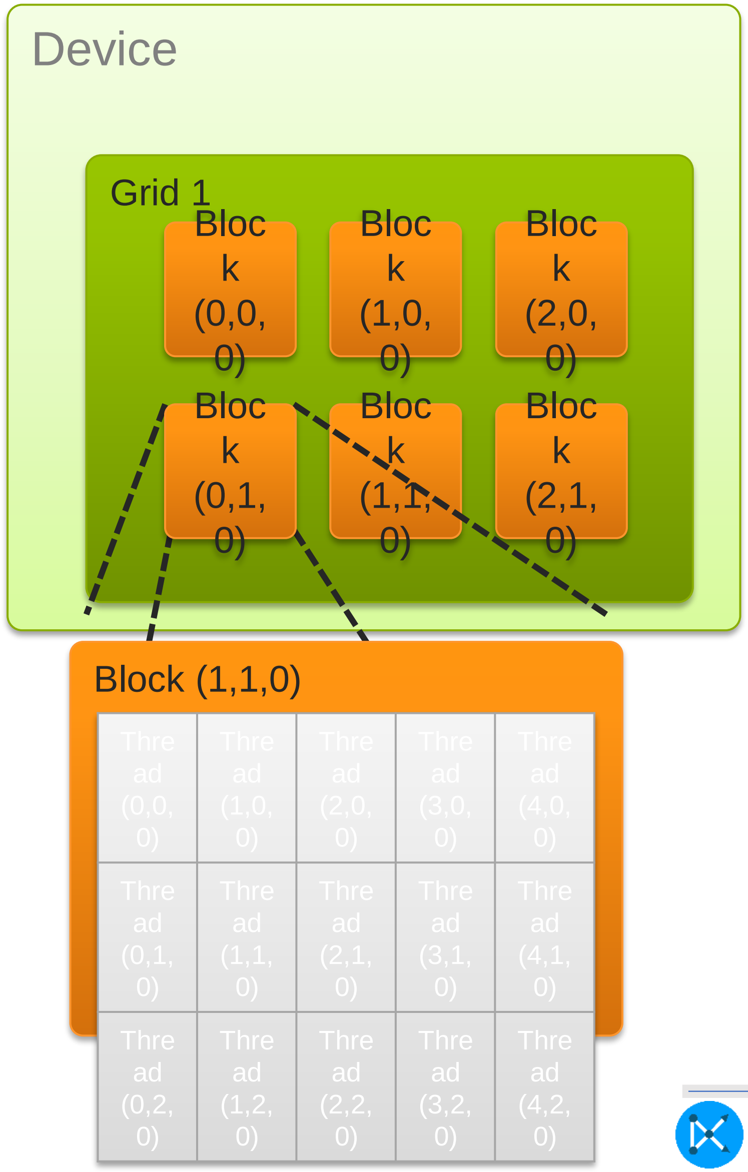
1. https://www.nvidia.com/enus/geforce/technologies/cuda/faq/
2. https://developer.nvidia.com/blog/analysis-driven-optimization-preparing-for-analysis-with-nvidia-nsight-compute-part-1/?ncid=so-nvsh-46518#cid=hpc06_so-nvsh_en-us
3. https://developer.nvidia.com/cuda-downloads.
4. https://developer.nvidia.com/higher-education-and-research

   Docs.nvidia.com/hpc-sdk/hpc-sdk-install-guide/index.html

IDs and Dimensions

- A kernel is launched as a grid of blocks of threads
  - `blockIdx` and `threadIdx` are 3D
  - We showed only one dimension (`x`)

- Built-in variables:
  - `threadIdx`
  - `blockIdx`
  - `blockDim`
  - `gridDim`



Additional Content

NVIDIA HPC SDK

NVIDIA HPC SDK, a comprehensive, integrated suite of compilers, libraries and tools for the NVIDIA HPC Platform

https://www.youtube.com/watch?v=hYoebR5HXmQ

NVIDIA HPC SDK, an integrated suite of compilers, libraries and tools for the NVIDIA HPC Platform with new developments that continue to open GPU computing to a wider audience of developers and users, including automatic acceleration and tensor core programmability in standard languages and novel libraries for compute and communication.

https://www.youtube.com/watch?v=COjvWNpxnxc

GPUs with NVIDIA CUDA architecture are usually programmed using the C language, but NVIDIA also provides a method of programming GPUS with Fortran. NVIDIA CUDA Fortran is distributed for free as part of NVIDIA HPC SDK (Software Development Kit) and is a set of extensions which permit CUDA calls inside Fortran. This seminar, aimed at Fortran programmers, will provide an introduction to GPU programming in Fortran, showing how to convert existing Fortran codes to use GPU acceleration.

Is it hard to write program for GPU?

- An algorithm developed for the CUDA architecture is actually a serial algorithm that can be run on many different processors simultaneously, often called a kernel.
- GPU takes this kernel and executes it in parallel by launching thousands of instances across many processors in the GPU.
- Since most algorithms start off as serial algorithms, it's often trivial to port programs to the CUDA architecture.

There's no need to completely re-architect the entire program to be multi-threaded as you do with modern multi-core CPUs.

https://www.nvidia.com/en-us/geforce/technologies/cuda/faq/

CUDA compatibility

The CUDA Driver API and the CUDA Runtime are programming interfaces to CUDA. Their version number enables developers to check the features associated with these APIs, and decide whether an application requires a newer (later) version than the one currently installed.

For example:

A CUDA Driver version 1.1 will run an application (or plugins, and libraries including the CUDA Runtime) compiled for it, and will also run the same application compiled for the earlier version, for example, version 1.0, of the CUDA Driver. That is to say, the CUDA Driver API is backward compatible.

However, a CUDA Driver version 1.1 will not be able to run an application that was compiled for the later version, for example, version 2.0, of the CUDA Driver. The CUDA Driver API is not forward compatible.

Specific versions

To target specific versions of NVIDIA hardware and CUDA software, use the `-arch`, `-code`, and `-gencode` options of `nvcc`.

SIMT

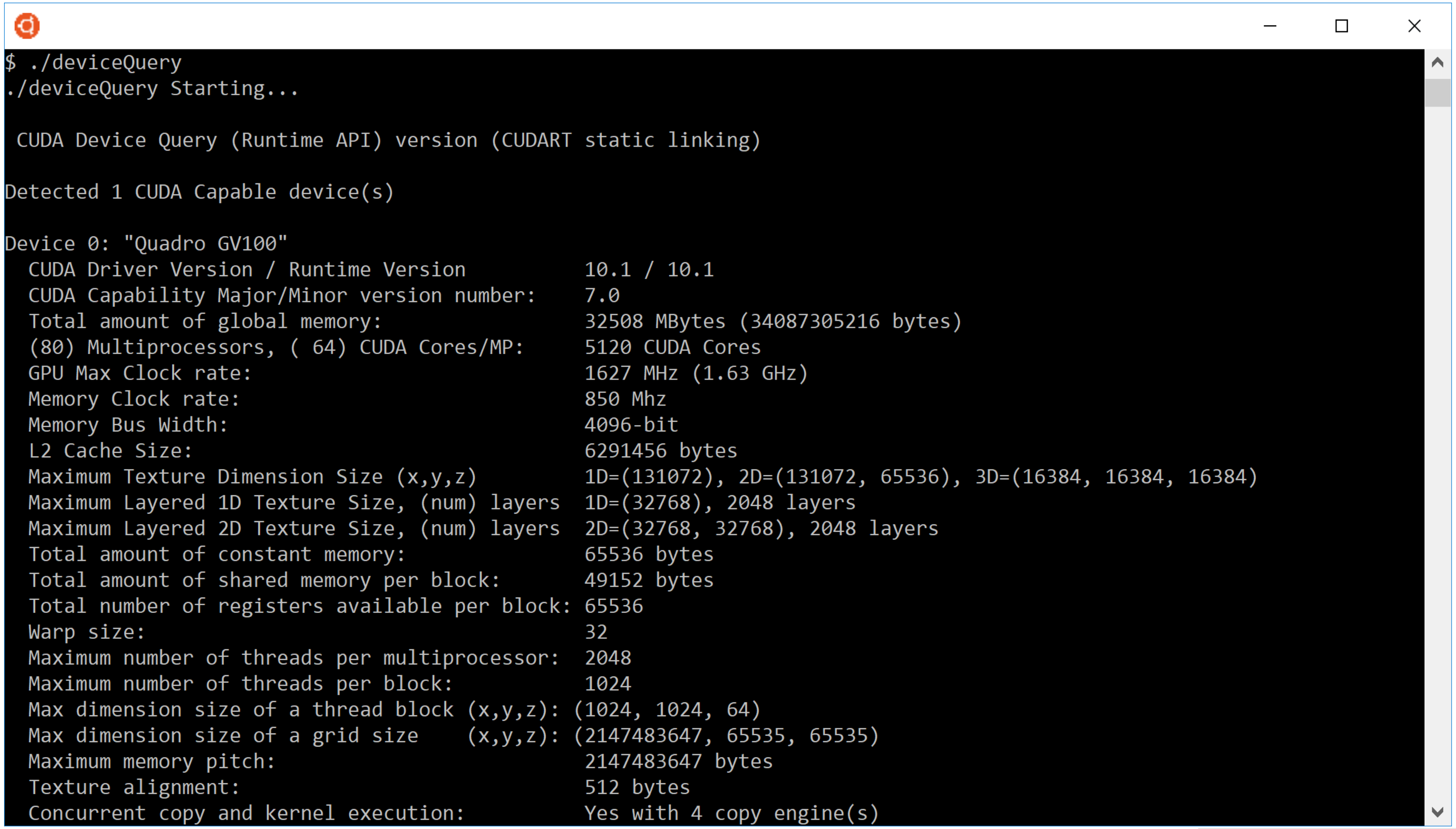
Single Instruction Multiple Thread - SIMT

nvcc compiler switches

nvcc compiler driver converts `.cu` files into C++ for the host system and CUDA assembly or binary instructions for the device. following are useful for optimization

- `-maxrregcount=N` specifies the maximum number of registers kernels can use at a per-file level. See Register Pressure.
  `--ptxas-options=-v` or `-Xptxas=-v` lists per-kernel register, shared, and constant memory usage.
- `-ftz=true` (denormalized numbers are flushed to zero)
- `-prec-div=false` (less precise division)
- `-prec-sqrt=false` (less precise square root)
- `-use_fast_math` compiler option of `nvcc` coerces every `functionName()` call to the equivalent `__functionName()` call. This makes the code run faster

The output deviceQuery



```
$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Quadro GV100"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    7.0
  Total amount of global memory:                 32508 MBytes (34087305216 bytes)
  (80) Multiprocessors, ( 64) CUDA Cores/MP:     5120 CUDA Cores
  GPU Max Clock rate:                            1627 MHz (1.63 GHz)
  Memory Clock rate:                             850 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 6291456 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 4 copy engine(s)
```

CUDA Driver Toolkit

- Running a CUDA application requires
  - the system with at least one CUDA capable GPU
  - a driver that is compatible with the CUDA Toolkit
- Each release of the CUDA Toolkit requires
  - a minimum version of the CUDA driver.
    - The CUDA driver is backward compatible
    - applications compiled against a particular version of the CUDA will continue to work on subsequent (later) driver releases.

CUDA Toolkit and Minimum Compatible Driver Versions

CUDA Toolkit and Minimum Compatible Driver Versions

CUDA Toolkit	Linux x86_64 Driver Version	Windows x86_64 Driver Version
CUDA 11.2.1 Update 1	>=460.32.03	>=461.09
CUDA 11.2.0 GA	>=460.27.03	>=460.82
CUDA 11.1.1 Update 1	>=455.32	>=456.81
CUDA 11.1 GA	>=455.23	>=456.38
CUDA 11.0.3 Update 1	>= 450.51.06	>= 451.82
CUDA 11.0.2 GA	>= 450.51.05	>= 451.48
CUDA 11.0.1 RC	>= 450.36.06	>= 451.22
CUDA 10.2.89	>= 440.33	>= 441.22
CUDA 10.1 (10.1.105 general release, and updates)	>= 418.39	>= 418.96
CUDA 10.0.130	>= 410.48	>= 411.31
CUDA 9.2 (9.2.148 Update 1)	>= 396.37	>= 398.26
CUDA 9.2 (9.2.88)	>= 396.26	>= 397.44
CUDA 9.1 (9.1.85)	>= 390.46	>= 391.29
CUDA 9.0 (9.0.76)	>= 384.81	>= 385.54
CUDA 8.0 (8.0.61 GA2)	>= 375.26	>= 376.51

CUDA Application Ecosystem

Optimized for HPC / AI

- New versions of deep learning frameworks such as Caffe2, MXNet, CNTK, TensorFlow, and others harness the performance of Volta to deliver faster training times and higher multi-node training performance.
- GPU accelerated libraries such as cuDNN, cuBLAS, and TensorRT leverage the new features of the Volta GV100 architecture to deliver higher performance for both deep learning inference and HPC applications.

CUDA Toolkit

- CUDA Compilers
- Nvcc
- CUDA Tools

CUDA Tools

- A preview version of a new tool, cu++filt, is included in this release.
    - NVCC produces mangled names, appearing in PTX files, which do not strictly follow the mangling conventions of the Itanium ABI--and are thus not properly demangled by standard tools such as binutils' c++filt.
- The new cu++filt utility will demangle all of these correctly.

CUDA Tools

NVCC

This is a reference document for nvcc, the CUDA compiler driver. nvcc accepts a range of conventional compiler options, such as for defining macros and include/library paths, and for steering the compilation process.

CUDA-GDB

The NVIDIA tool for debugging CUDA applications running on Linux and QNX, providing developers with a mechanism for debugging CUDA applications running on actual hardware. CUDA-GDB is an extension to the x86-64 port of GDB, the GNU Project debugger.

CUDA-MEMCHECK

CUDA-MEMCHECK is a suite of run time tools capable of precisely detecting out of bounds and misaligned memory access errors, checking device allocation leaks, reporting hardware errors and identifying shared memory data access hazards.

Compute Sanitizer

The user guide for Compute Sanitizer.

CUDA Tools

Nsight Eclipse Plugins Installation Guide

Nsight Eclipse Plugins Installation Guide

Nsight Eclipse Plugins Edition

Nsight Eclipse Plugins Edition getting started guide

Nsight Compute

The NVIDIA Nsight Compute is the next-generation interactive kernel profiler for CUDA applications. It provides detailed performance metrics and API debugging via a user interface and command line tool.

Profiler

This is the guide to the Profiler.

CUDA Binary Utilities

CUDA Developer Tools

- nvprof and Visual Profiler
- CUPTI
- Nsight Compute Profiler
  - The Nsight Compute can collect a large range of data on kernel execution
  - A *rule* in Nsight Compute is a set of instructions to the profiler that indicate what metrics are to be gathered and how they are to be displayed or interpreted
  - Make use of rules embedded in the analysis output from Nsight Compute

https://developer.nvidia.com/blog/analysis-driven-optimization-preparing-for-analysis-with-nvidia-nsight-compute-part-1/?ncid=so-nvsh-46518#cid=hpc06_so-nvsh_en-us

Nsight Compute

- Nsight Compute - *continued*
  - using Nsight Compute 2020.2 with appropriate path setup, type **ncu-ui**. When the initial dialog box opens, choose **Quick Launch**, **Continue**

Nsight Compute

[Application Note](#)

[CUDA for Tegra](#)

This application note provides an overview of NVIDIA® Tegra® memory architecture and considerations for porting code from a discrete GPU (dGPU) attached to an x86 system to the Tegra® integrated GPU (iGPU).

https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

Introductory and Advanced Accelerated Computing (typical agenda by NVIDIA)

- Introduction to CUDA C
- Memory and Data Locality
- Thread Execution Efficiency
- Memory Access Performance
- Parallel Computation Patterns
- Efficient Host-Device Data Transfer
- OpenACC, MPI, OpenCL
- Unified Memory
- Dynamic Parallelism
- Multi-GPU Systems
- CUDA Library Usage

Accelerated computing

CPU
Optimized for Serial Tasks

CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

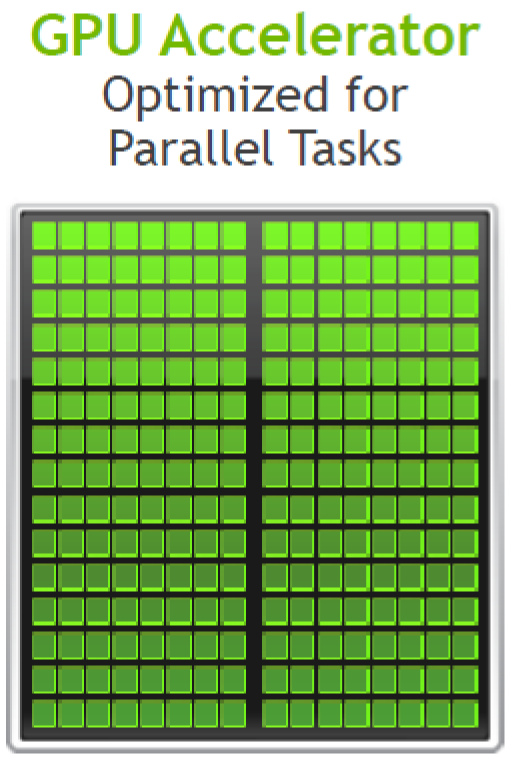
Accelerated computing

GPU Strengths

- High bandwidth main memory
- Latency tolerant via parallelism
- Significantly more compute resources
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance



DATAEVER CONSULTING

Sample C CUDA code

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    …
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Streaming Multiprocessors and caches

- Caches:
  - L1 cache. (for reducing memory access latency).
  - Shared memory. (for shared data between threads).
  - Constant cache (for broadcasting of reads from a read-only memory).
  - Texture cache. (for aggregating bandwidth from texture memory).
  - Schedulers for warps. (these are for issuing instructions to warps based on particular scheduling policies).
  - A substantial number of registers. (an SM may be running a large number of active threads at a time, so it is a must to have registers in thousands.)

Tensor core and CUDA core

CUDA cores:

Does a single value multiplication per one GPU clock

1 x 1 per GPU clock

TENSOR cores:

Does a matrix multiplication per one GPU clock

mat [A] x mat [B] per one GPU clock

To be more precise TENSOR core does the computation of many CUDA cores in the same time

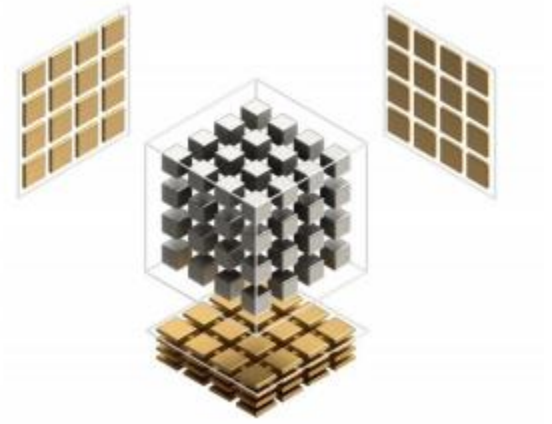
Tensor core and CUDA core

Tesla V100 and Titan V have tensor cores.

- Both GPUs have 5120 cuda cores where each core can perform up to
  - 1 single precision multiply-accumulate operation (e.g. in fp32: x += y * z) per 1 GPU clock
- Each Tensor core perform operations on small matrices with size 4x4.
  - Each tensor core can perform 1 matrix multiply-accumulate operation per 1 GPU clock.
  - It multiplies two fp16 matrices 4x4 and adds the multiplication product fp32 matrix (size: 4x4) to accumulator (that is also fp32 4x4 matrix).
- It is called mixed precision because input matrices are fp16 but multiplication result and accumulator are fp32 matrices.
- Probably, the proper name would be just 4x4 matrix cores
  - however NVIDIA decided to use "tensor cores".

https://www.techspot.com/article/2049-what-are-tensor-cores/

https://stackoverflow.com/questions/47335027/what-is-the-difference-between-cuda-vs-tensor-cores#:~:text=Tensor%20cores%20use%20a%20lot,changing%20the%20output%20that%20much.



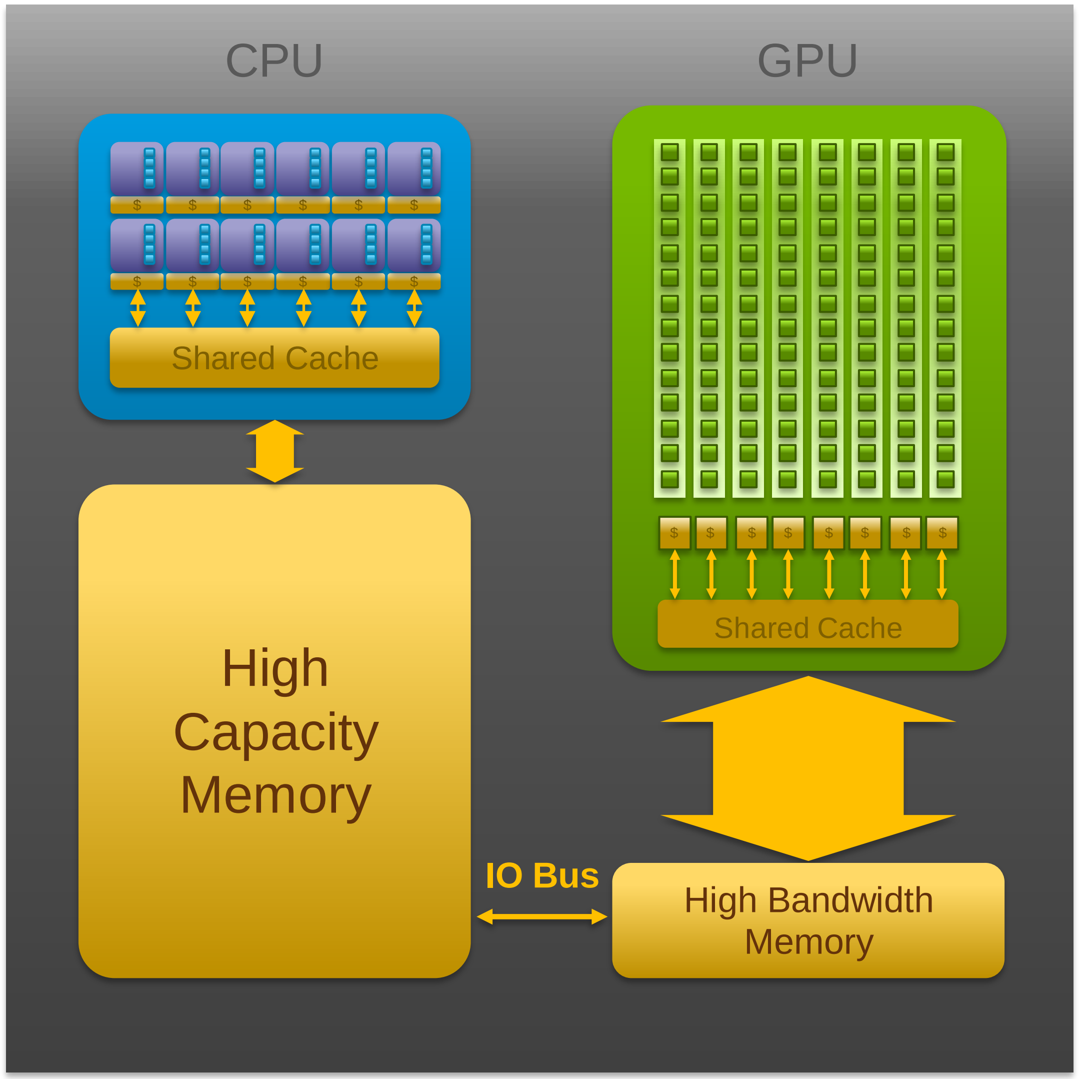
CUDA History

- In November 2006, NVIDIA introduced CUDA
- CUDA is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs

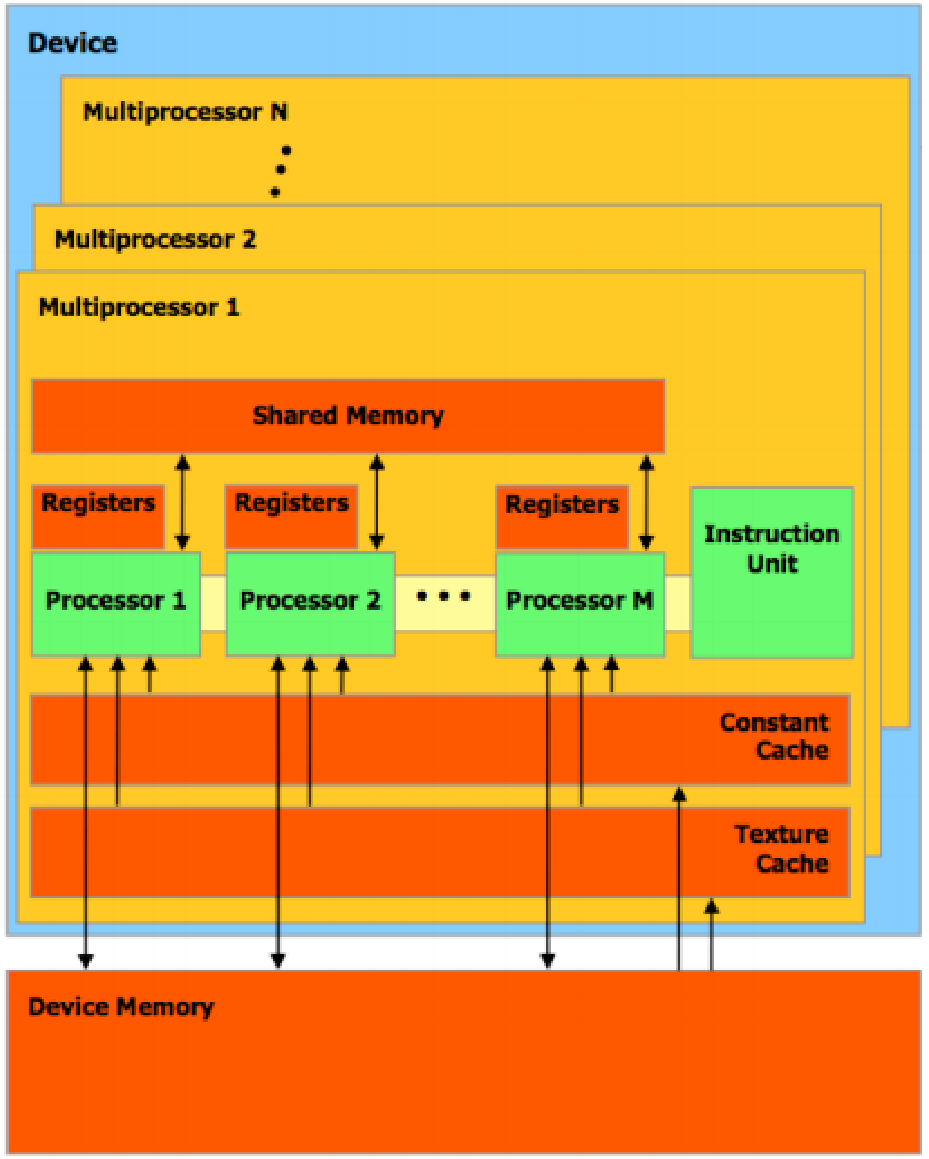
CPU + GPU

Physical Diagram

- CPU memory is larger, GPU memory has more bandwidth
- CPU and GPU memory are usually separate, connected by an I/O bus (traditionally PCI-e)
- Any data transferred between the CPU and GPU will be handled by the I/O Bus
- The I/O Bus is relatively slow compared to memory bandwidth
- The GPU cannot perform computation until the data is within its memory



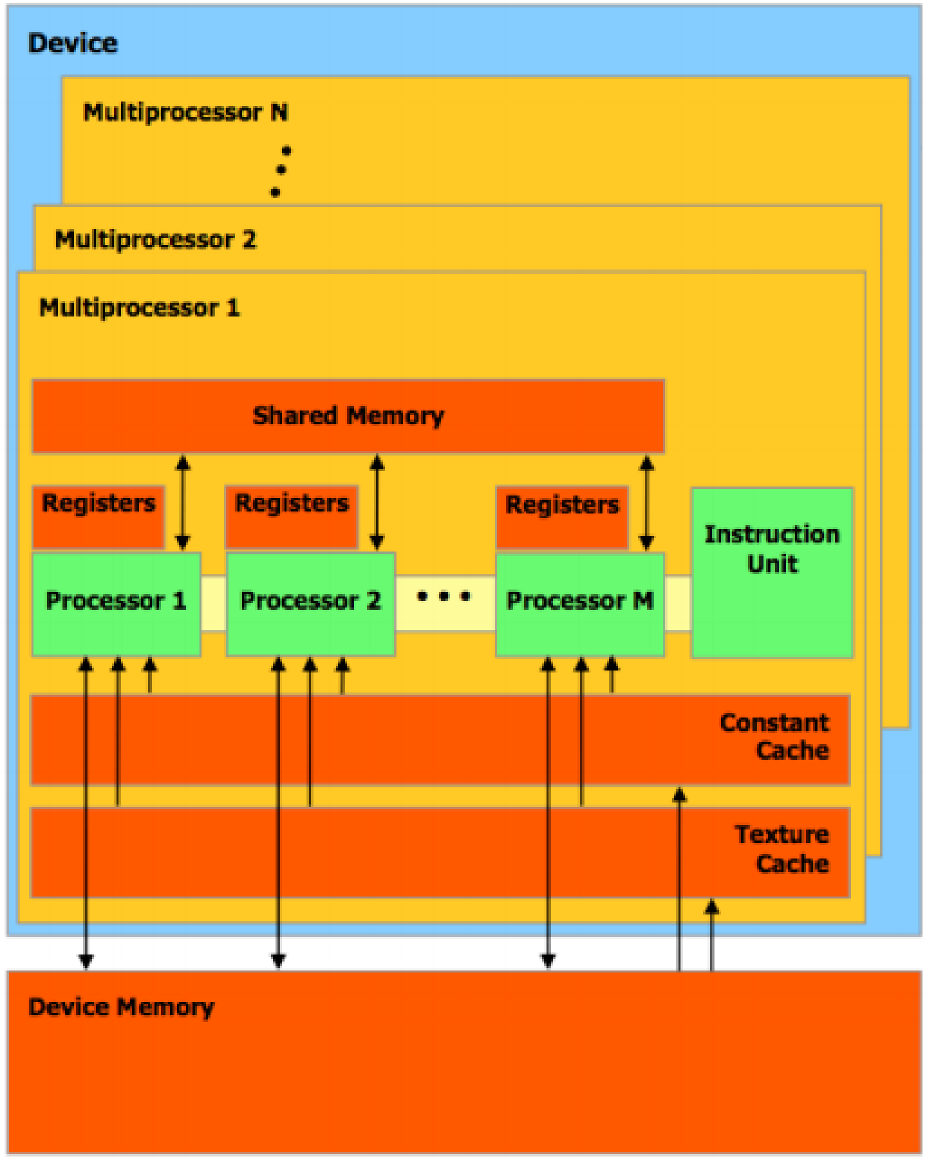
CUDA thread synchronization



- CUDA provides a means to synchronize threads within a thread -block to ensure that all threads reach certain point in execution before making further progress

DATAEVER CONSULTING

CUDA memory types



- Shared memory partitioned amongst Thread Blocks resident on the Streaming Multiprocessors
- Registers are partitioned amongst Threads

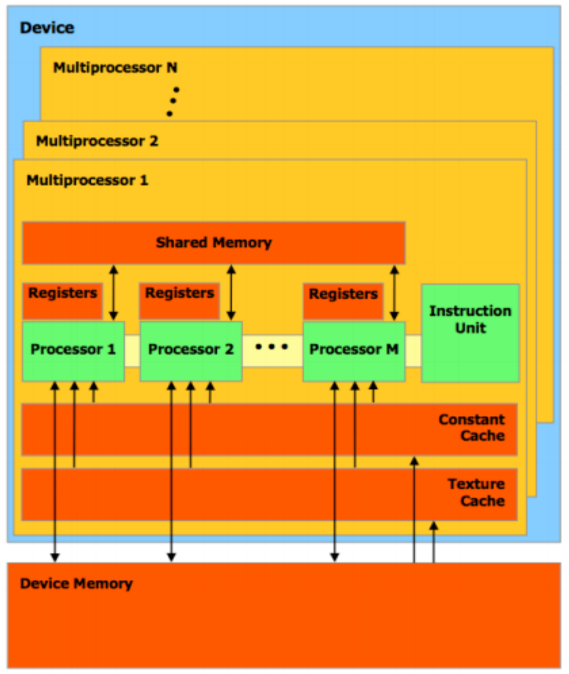
Types of Memory

- Data stored in register memory is visible only to the thread that wrote it and lasts only for the lifetime of that thread

- Local memory has the same scope rules as register memory, but performs slower

- Data stored in shared memory is visible to all threads within that block and lasts for the duration of the block. This is invaluable because this type of memory allows for threads to communicate and share data between one another

- Data stored in global memory is visible to all threads within the application (including the host), and lasts for the duration of the host allocation compared to global memory



Types of Memory

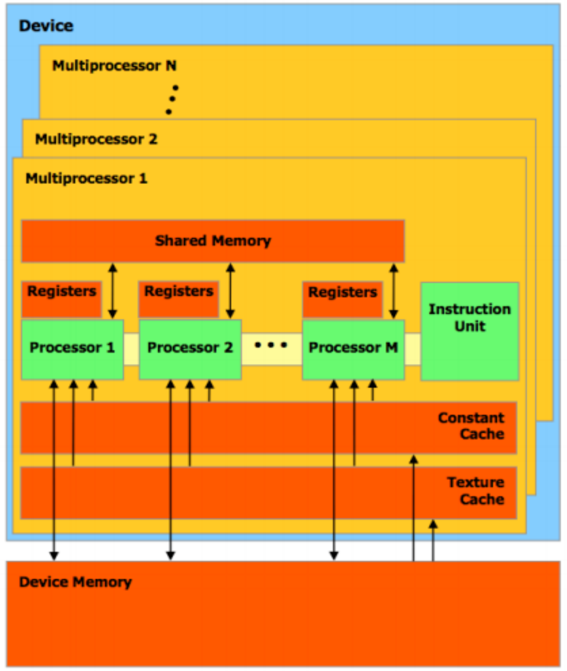
Constant and texture memory for special applications

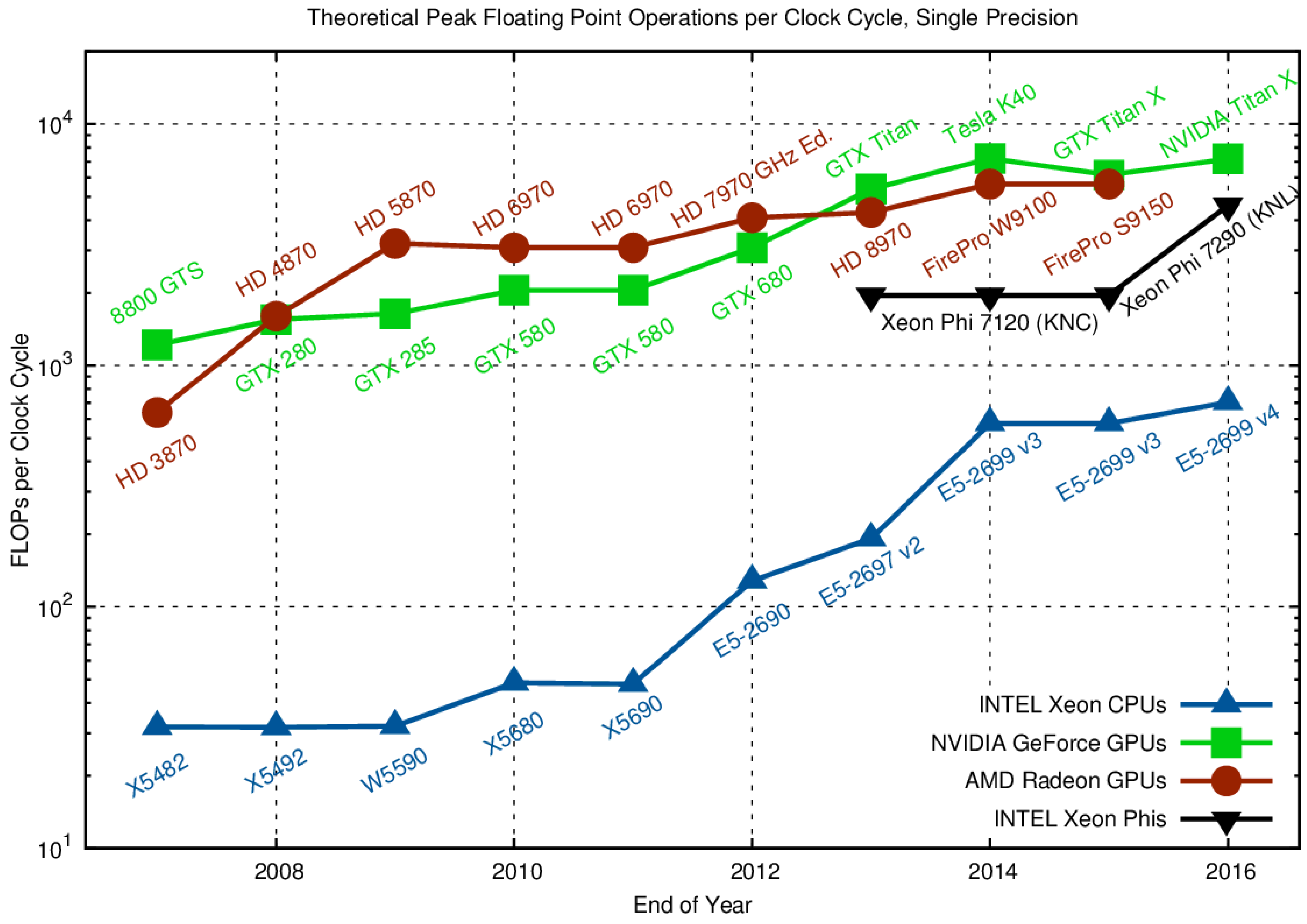
Constant memory is used for data that will not change over the course of a kernel execution and is read only

- Using constant rather than global memory can reduce the required memory bandwidth, however, this performance gain can only be realized when a warp of threads read the same location.

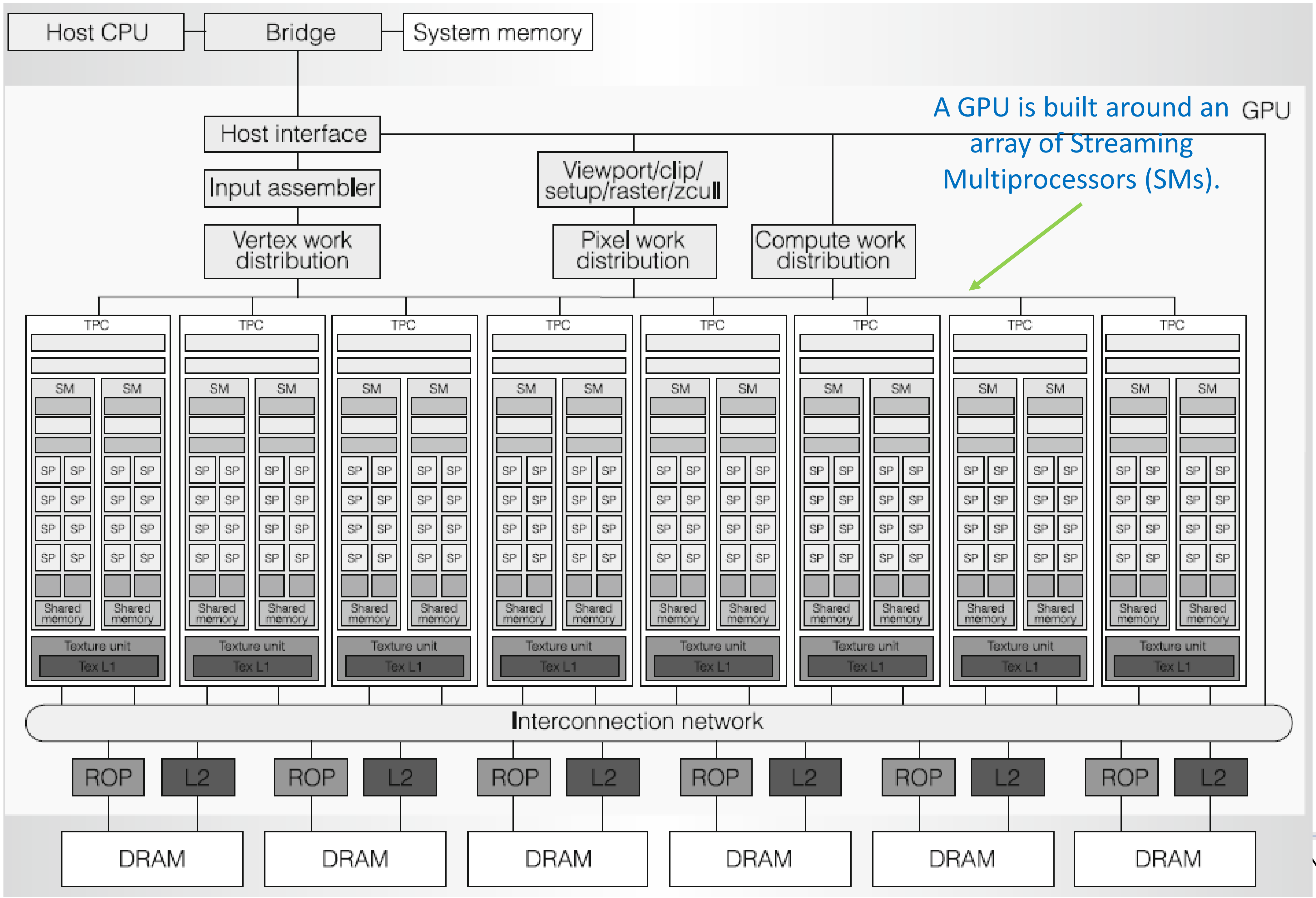
Texture memory is another variety of read-only memory on the device

- When all reads in a warp are physically adjacent, using texture memory can reduce memory traffic and increase performance compared to global memory.

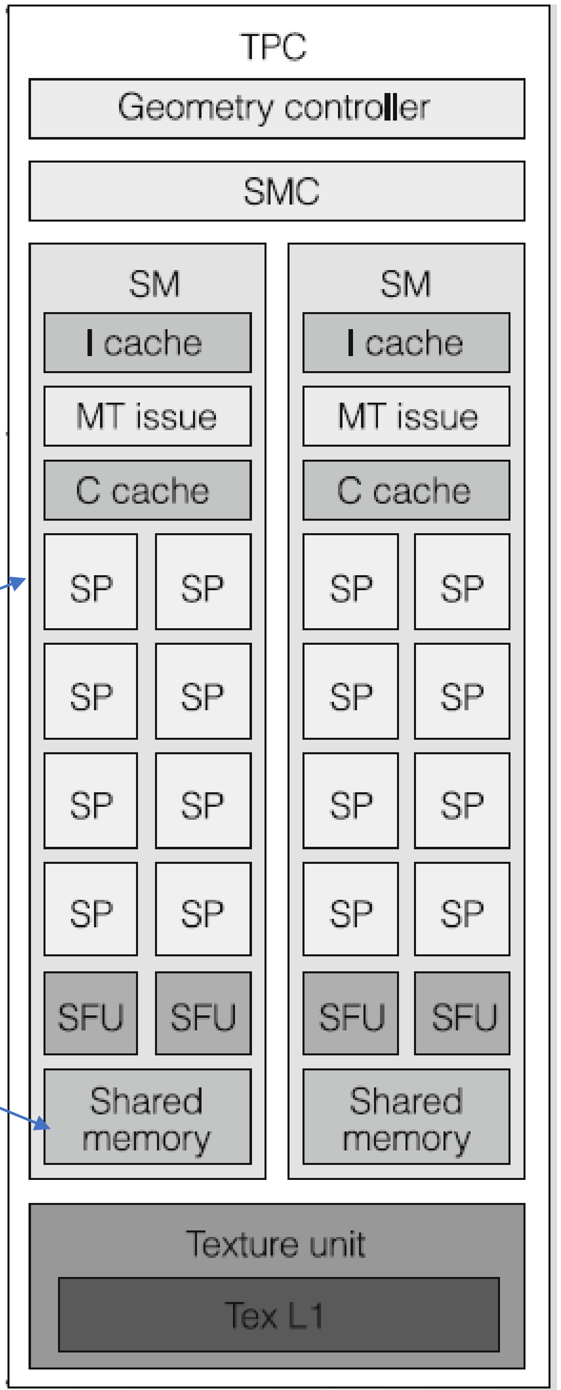




DATAEVER CONSULTING

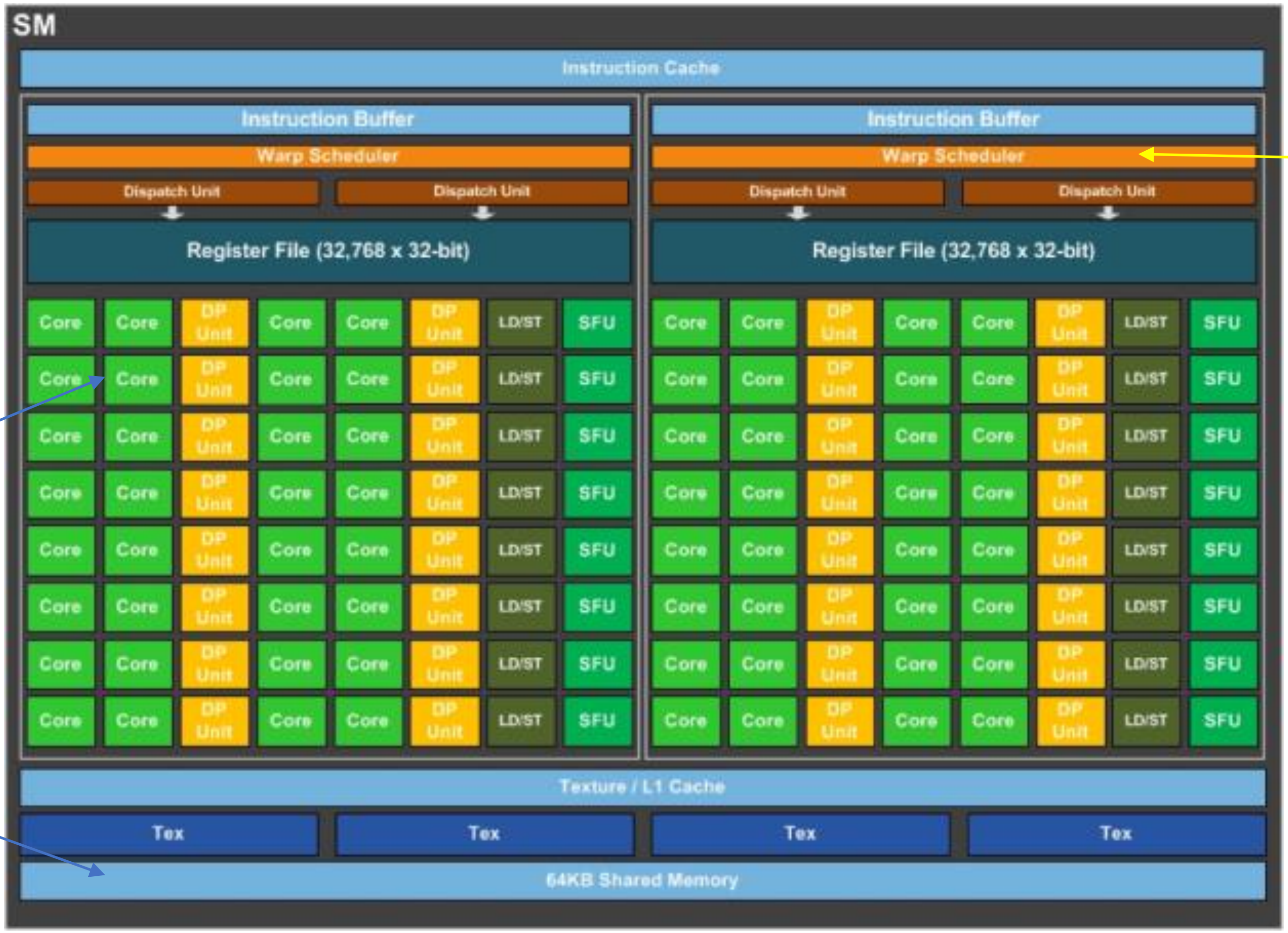


A GPU is built around an array of Streaming Multiprocessors (SMs).



Threads from the same block have access to a shared memory(SM) and their execution can be synchronized

CUDA-Compute Unified Device Architecture
SIMT – Single Instruction Multiple Threads
TPC- Texture Processing Cluster
GPC – GPU Processing Cluster
SM - Streaming Multi Processor
SMC – SM cluster
SFU – SP Function Unit
SP – Core / sequential processor



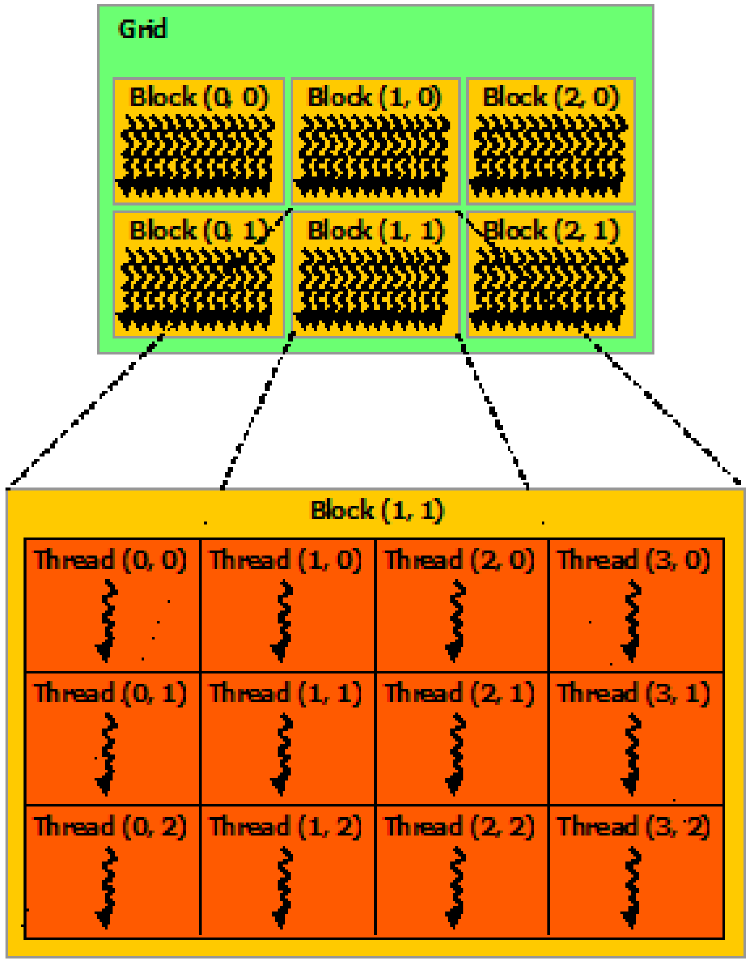
A warp is a collection of threads, 32

Threads from the same block have access to a shared memory(SM) and their execution can be synchronized

SM - Streaming Multi Processor
SFU – SP Function Unit
SP – Core / sequential processor

Grid of Thread blocks

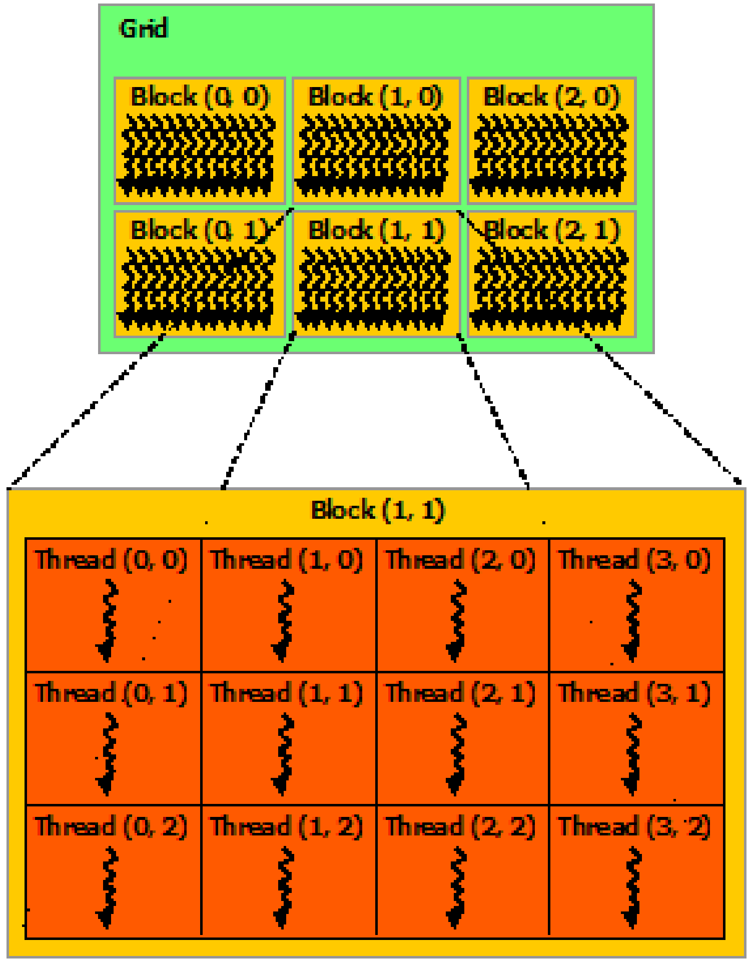
two dimensional



- Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks
- The number of thread blocks in a grid is usually dictated by the size of the data being processed, which typically exceeds the number of processors in the system
- The number of threads per block and the number of blocks per grid specified in the `<<<...>>>` syntax can be of type `int` or `dim3`

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Grid of Thread blocks two dimensional



Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in `blockIdx` variable.

The dimension of the thread block is accessible within the kernel through the built-in `blockDim` variable.

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- Let’s take a look at main()…

Addition on the Device: `add()`

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        *c = *a + *b;
}
```

- Let’s take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                  // host copies of a, b, c
        int *d_a, *d_b, *d_c;         // device copies of a, b, c
        int size = 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);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

```
        // Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Tensor core and CUDA core

CUDA core

- Each individual CUDA core can perform
- one calculation per revolution of the GPU.

Tensor core

- Tensor Cores are specialized execution units designed specifically for performing the tensor/matrix operations that are the core compute function used in Deep Learning.

- Tensor cores, can calculate entire 4x4 matrix operation in a clock.

https://towardsdatascience.com/what-on-earth-is-a-tensorcore-bad6208a3c62

CUDA Parallel Computing Platform

Programming Approaches

Libraries	OpenACC Directives	CUDA Programming
"Drop-in" Acceleration	Easily Accelerate Apps	Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler Tool Chain

LLVM COMPILER INFRASTRUCTURE

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

CUDA Software Environment

- CUDA comes with a software environment that allows developers to use C, C++ as high-level programming languages.
- other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC, Open MP

Profiling the code for CUDA activities

```
cd /home/sambath/WORK-FIREFLY/HANDS-ON-LAB/CUDAC/EX14-JACOBI/solution

nvprof ./cfd

vi *.txt
```

CUDA Software Environment

- The advent of multicore CPUs and GPUs means that processor chips are now parallel systems
- The challenge is to develop application software that scales up to leverage the many processor cores, and 3D graphics applications scales leverage many GPUs
- The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- To look closer, we need a new example…