



GPU Parallel Programming: CUDA

National Tsing-Hua University
2019, Summer Semester



Outline

- CUDA Basic
- Example Code Study
 - Hello world
 - All pairs shortest path
- Time Measurement & Debug API
- Multi-GPUs
- Dynamic Parallelism

What is CUDA?

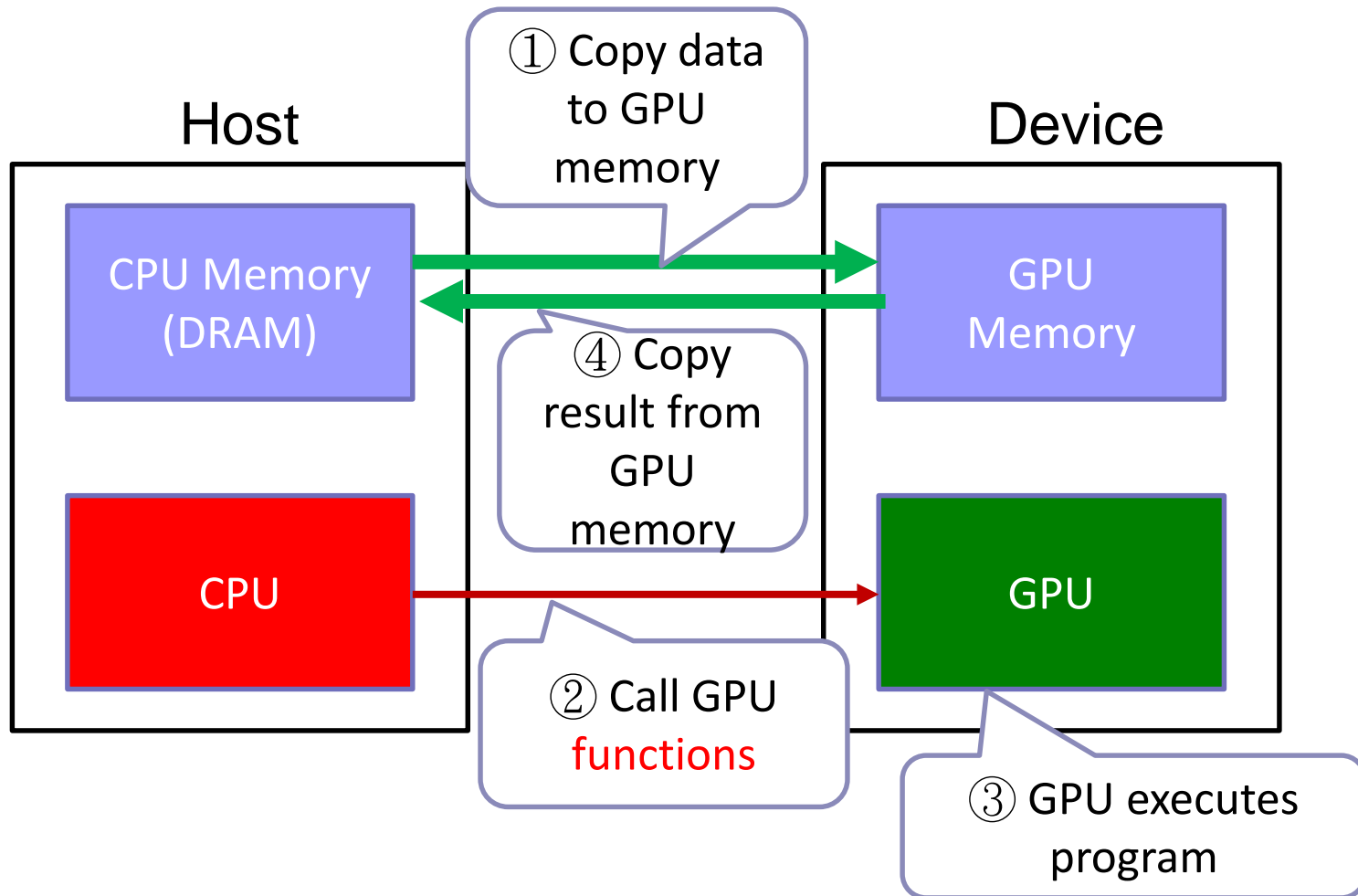
■ CUDA: Compute Unified Device Architecture

- CUDA is a **compiler** and **toolkit** for programming NVIDIA GPUs
- Enable heterogeneous computing and horsepower of GPUs
- CUDA API extends the C/C++ programming language
- Express SIMD parallelism
- Give a high level abstraction from hardware

■ CUDA version

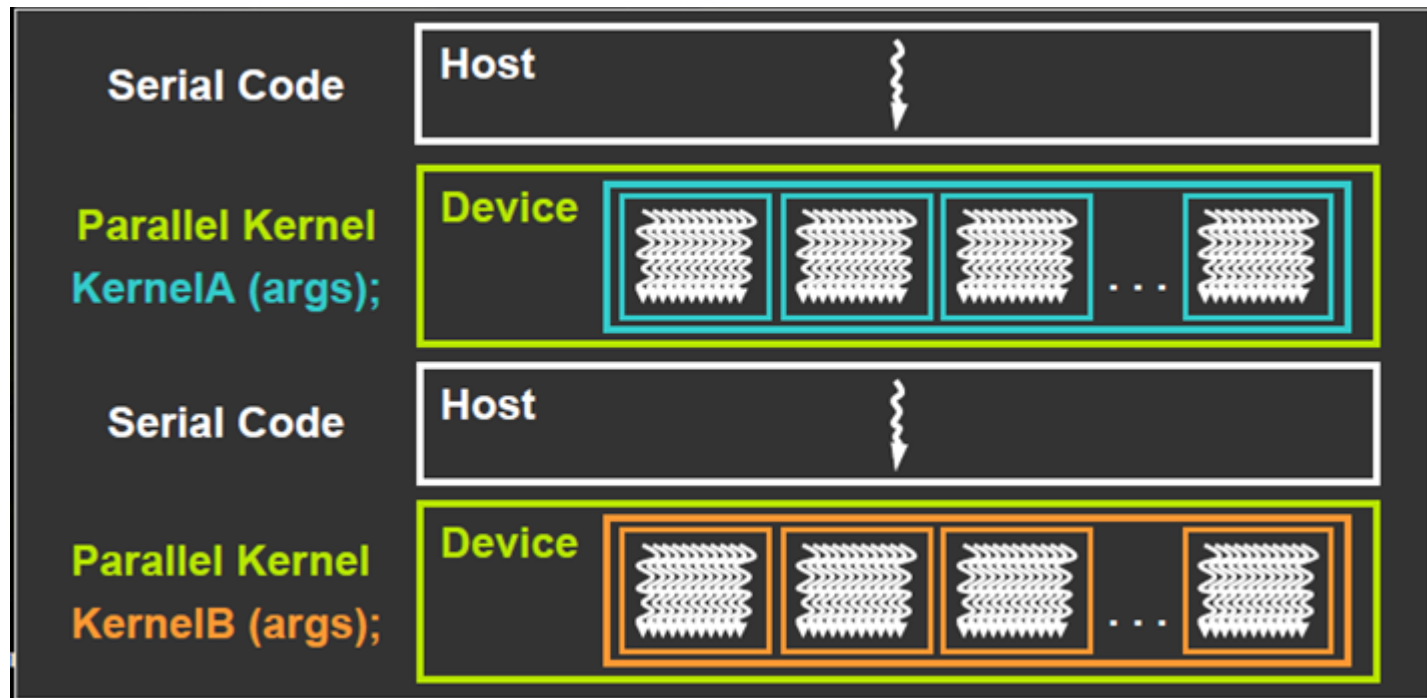
- The latest version is 7.5
- We will learn the syntax of 4.2 first, because we will use it in the optimization lectures

CUDA program flow



CUDA Programming Model

- CUDA = serial program with parallel kernels, all in C
 - Serial C code executes in a host thread (i.e. CPU thread)
 - Parallel kernel C code executes in many devices threads across multiple processing elements (i.e. GPU threads)



CUDA program framework

GPU code
(parallel)

CPU code
(serial or
parallel if
p-thread/
OpenMP/T
BB/MPI is
used.)

```
#include <cuda_runtime.h>
```

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

```
int main() {  
    ...  
    cudaMalloc(...)  
    cudaMemcpy(...)  
    ...  
    my_kernel<<<nblock,blocksize>>>(...)  
    ...  
    cudaMemcpy(...)  
    ...  
}
```

Kernel = Many Concurrent Threads

- One kernel is executed at a time on the device
- Many thread execute each kernel
 - Each thread executes the same code
 - ... on the different data based on its threadID

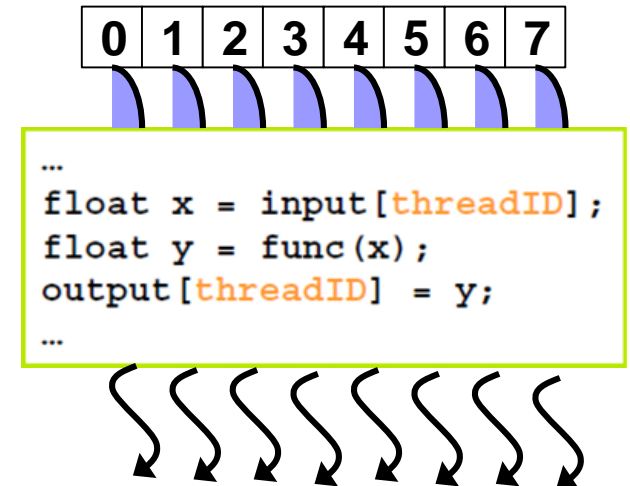
- CUDA thread might be

- Physical threads

- ◆ As on NVIDIA GPUs
 - ◆ GPU thread creation and context switching are essentially free

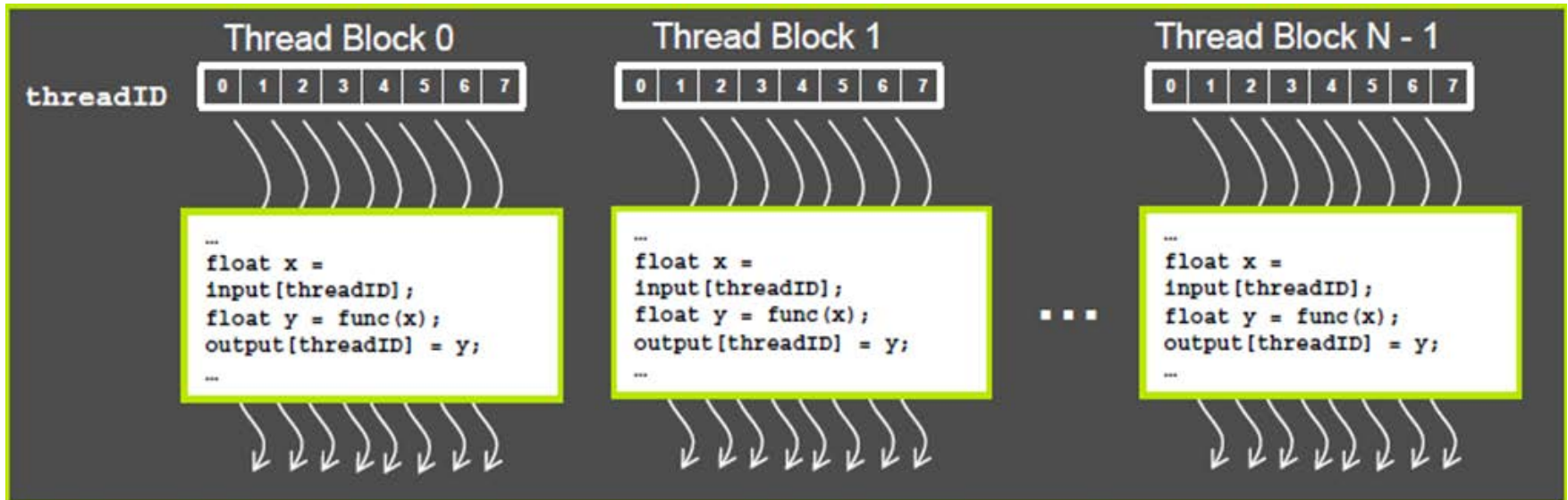
- Or virtual threads

- ◆ E.g. 1 CPU core might execute multiple CUDA threads



Hierarchy of Concurrent Threads

- Threads are grouped into thread blocks
 - Kernel = grid of thread blocks

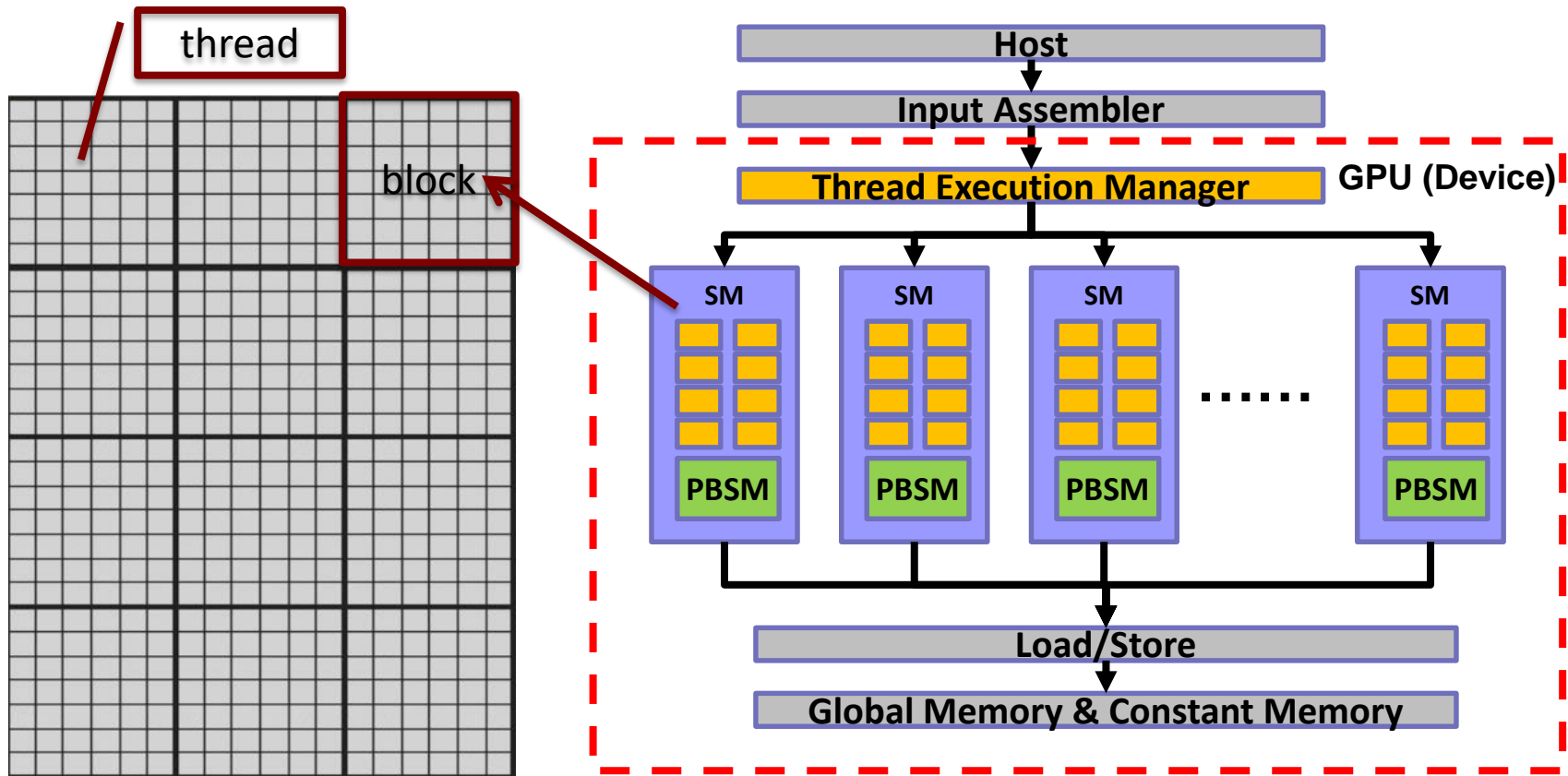


- By definition, threads in the same block may synchronized with barriers, but not between blocks

```
scratch[threadID] = begin[threadID];  
__syncthreads();  
int left = scratch[threadID - 1];
```

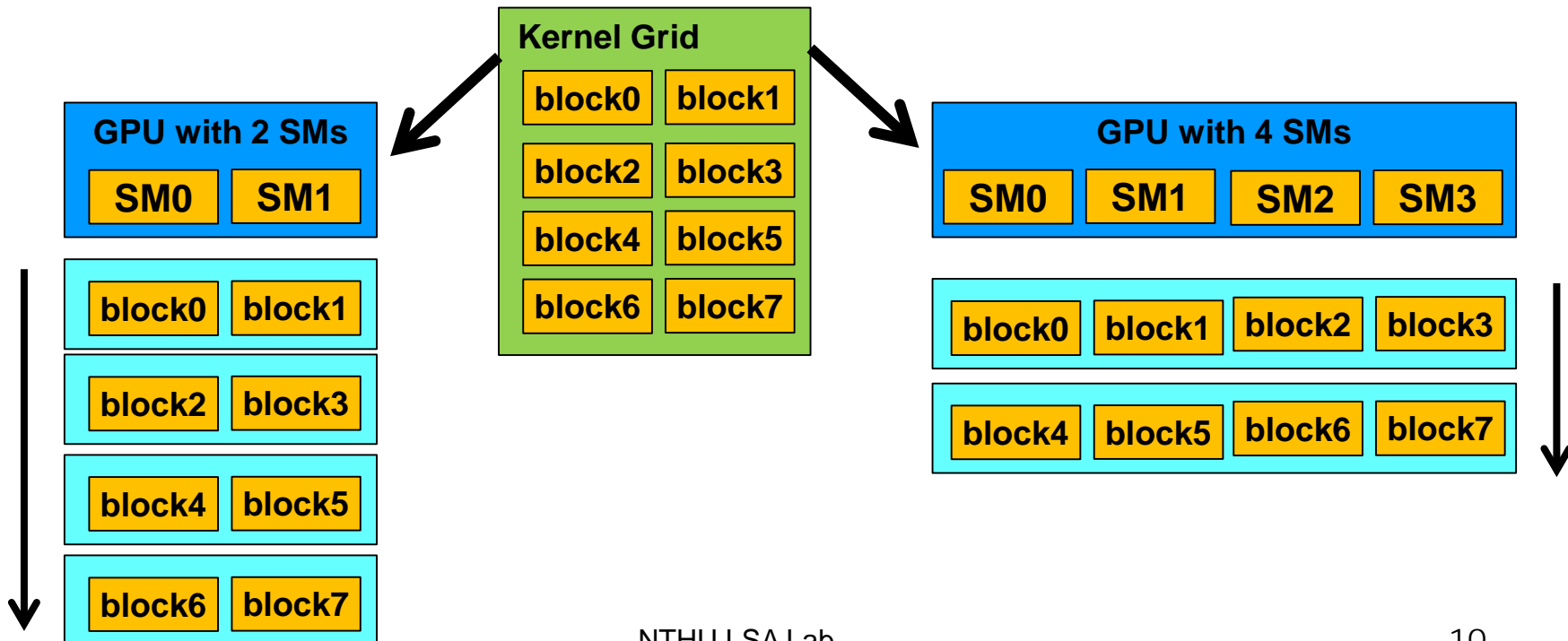

Software Mapping

- Software: grid → blocks → threads
- Hardware: GPU(device) → SM(multicore processor) → core



Transparent Scalability

- Thread blocks cannot synchronize
 - So they can run in any order, concurrently or sequentially
- This independence gives scalability:
 - A kernel scales across any number of parallel cores

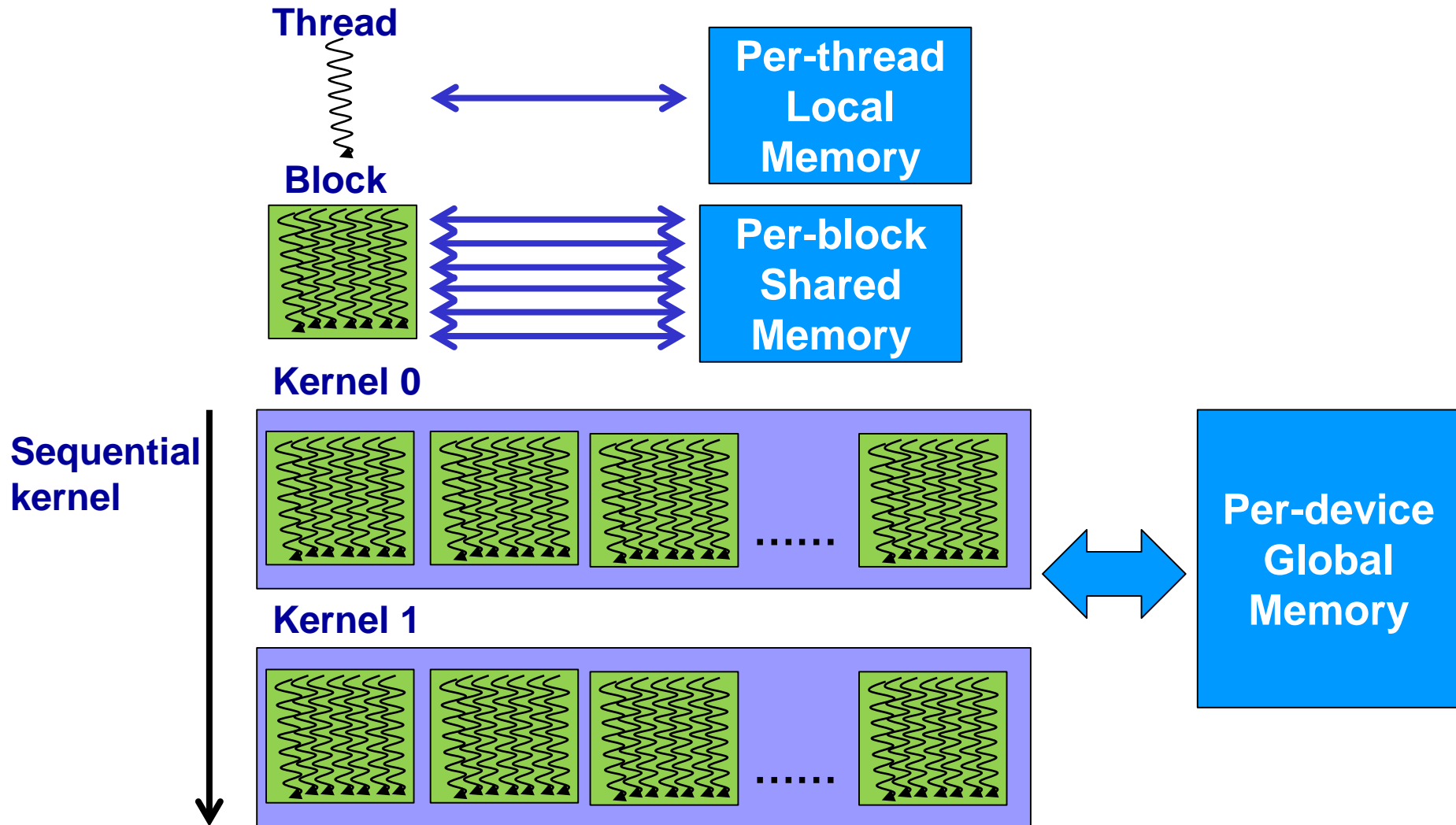


Thread group limits

- The maximum number of threads per block is limited
 - 512 before CUDA 2.0
 - 1024 after CUDA 2.0
- The maximum number of blocks is limited
 - 65535 before CUDA 3
 - $2^{31}-1$ after CUDA 3
- Total number of threads = threads per block * number of blocks

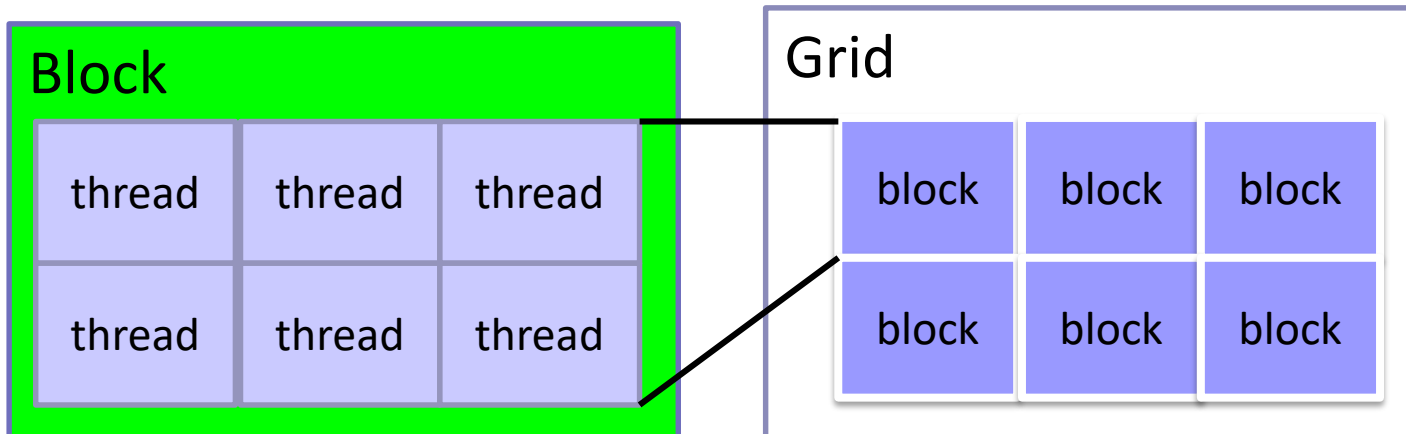
Use `deviceQuery.cpp` to find out your limits

Memory Hierarchy



CUDA Programming Terminology

- Host : CPU
- Device : GPU
- Kernel : functions executed on GPU
- Thread : the basic execution unit
- Block : a group of threads
- Grid : a group of blocks



Quiz

- What is a CUDA kernel?
- What are the 4 steps of CUDA program flow?
- Can a kernel run on two different devices at the same time?
- Can a kernel run across multiple SM processors?
- Can the threads from a same block run across multiple SM processors?
- Why shared block memory can only be accessed by the threads in the same blocks?
- Why `__syncthreads()` is not supported across blocks?

CUDA Language

Philosophy: provide minimal set of extensions necessary

■ Kernel launch

```
kernelFunc<<< nB, nT, nS, Sid >>>(...); // nS and Sid are optional
```

- nB : number of blocks per grid (grid size)
- nT : number of threads per block (block size)
- nS : shared memory size (in bytes)
- Sid : stream ID, default is 0

■ Build-in device variables

- threadIdx; blockIdx; blockDim; gridDim

■ Intrinsic functions that expose operations in kernel code

```
__syncthreads();
```

■ Declaration specifier to indicate where things live

```
__global__ void KernelFunc(...); // kernel function, run on device  
__device__ void GlobalVar;       // variable in device memory  
__shared__ void SharedVar;       // variable in per-block shared memory
```

Thread and Block IDs

- The index of threads and blocks is a 3-tuple: `dim3`

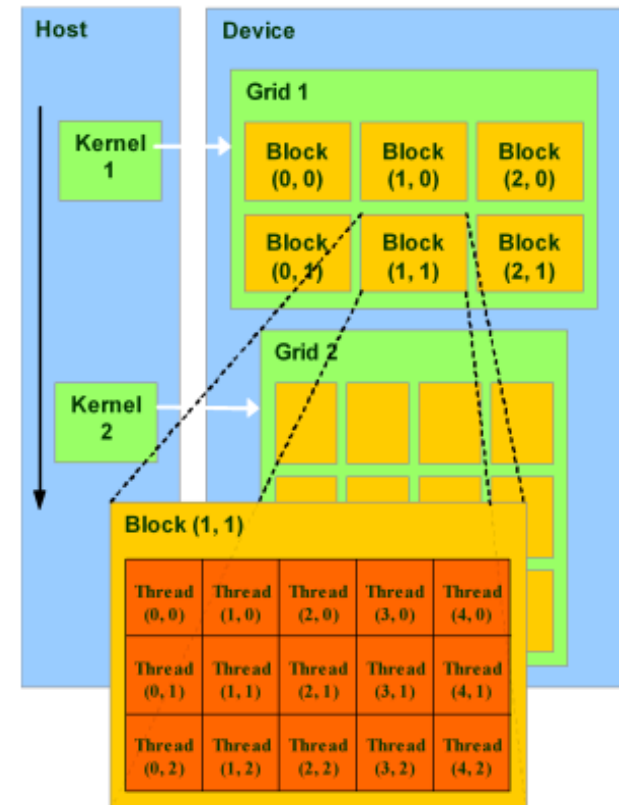
- `dim3` is a struct defined in `vector_types.h`

```
struct dim3 { x; y; z; };
```

- Example:

```
➤ dim3 grid(3, 2);  
➤ dim3 blk(5, 3);  
➤ my_kernel<<< grid, blk >>>();
```

- Each thread can be uniquely identified by a pair of index (x,y).



Function Qualifiers

Function qualifiers	limitations
<code>__device__</code> function	Executed on the device Callable from the device only
<code>__global__</code> function	Executed on the device Callable from the host only (must have void return type!)
<code>__host__</code> function	Executed on the host Callable from the host only
Functions without qualifiers	Compiled for the host only
<code>__host__ __device__</code> function	Compiled for both the host and the device

Variable Type Qualifiers

Variable qualifiers	limitations
<code>__device__ var</code>	<ul style="list-style-type: none">• Resides in device's global memory space
<code>__constant__ var</code>	<ul style="list-style-type: none">• Has the lifetime of an application• Is accessible from all the threads within the grid and from the host through the runtime library• Resides in device's constant memory space
<code>__shared__ var</code>	<ul style="list-style-type: none">• Resides in the shared memory space of a thread block• Has the lifetime of the block• Is only accessible from all the threads within the block

Device memory operations

■ Three functions:

- `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
- Similar to the C's `malloc()`, `free()`, `memcpy()`

1. `cudaMalloc(void **devPtr, size_t size)`

- `devPtr`: return the address of the allocated device memory
- `size`: the allocated memory size (**bytes**)

2. `cudaFree (void *devPtr)`

3. `cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)`

- `count`: size in **bytes** to copy

cudaMemcpyKind

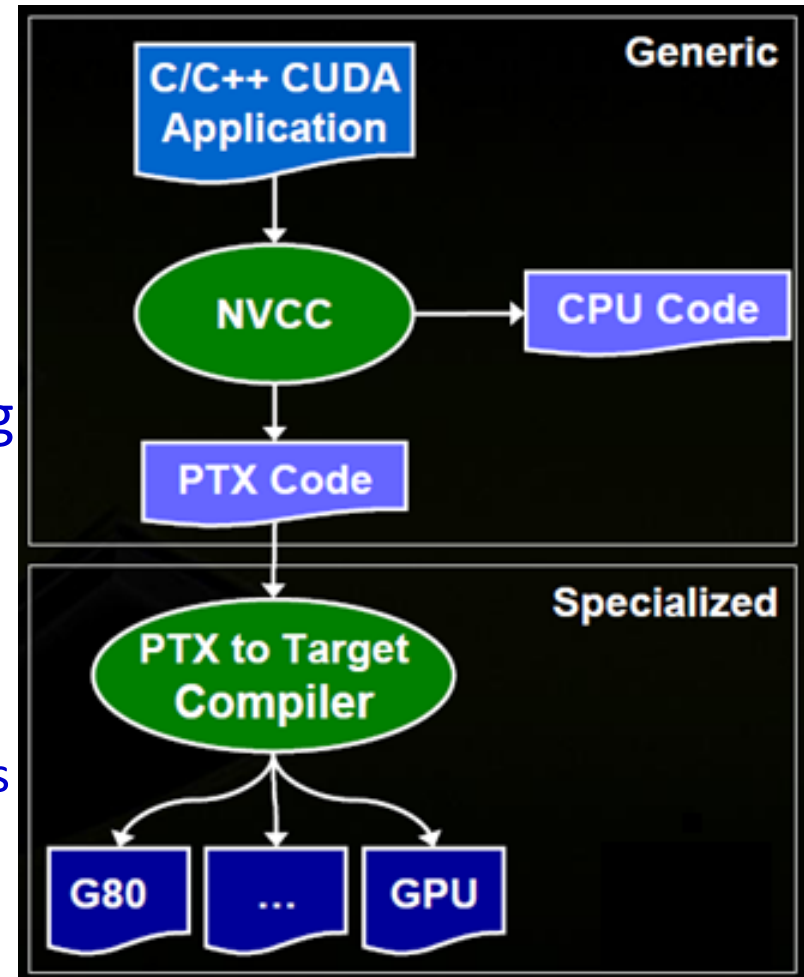
- one of the following four values

cudaMemcpyKind	Meaning	dst	src
cudaMemcpyHostToHost	Host → Host	host	host
cudaMemcpyHostToDevice	Host → Device	device	host
cudaMemcpyDeviceToHost	Device → Host	host	device
cudaMemcpyDeviceToDevice	Device → Device	device	device

host to host has the same effect as memcpy()

Program Compilation

- Any source file containing CUDA language must be compiled with NVCC
 - NVCC separates code running on the host from code running on the device
- Two-stage compilation:
 - Virtual ISA
 - ◆ PTX: Parallel Threads eXecutions
 - Device-specific binary object



Quiz

- How to index a 100 elements of an array under the following kernel launch setting?

```
// Kernel definition
__global__ void VecAdd(float* A)
{
    int i =           
    A[i] = A[i] + 1;
}
```

1. `my_kernel<<< 1, 100 >>>(A);`
2. `my_kernel<<< 100, 1 >>>(A);`
3. `my_kernel<<< 10, 10 >>>(A);`
4. `size=10; dim3 blk(size, size);`
`my_kernel<<< 1, blk >>>(A, size);`

Quiz

Function qualifiers	limitations
<code>__device__</code> function	
<code>__global__</code> function	
Functions without qualifiers	

Variable qualifiers	Limitations (Lifetime, Data Scope)
<code>__device__</code> var	
<code>__constant__</code> var	
<code>__shared__</code> var	

Outline

- CUDA Basic
- Example Code Study
 - Hello world
 - All pairs shortest path
- Time Measurement & Debug API
- Multi-GPUs
- Dynamic Parallelism

Example 1: Hello World!


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

■ Two new syntactic elements...

1. `__global__` indicates a function that runs on the device and is called from host code
2. `mykernel<<<1,1>>>()`;
Triple angle brackets mark a call from host code to device code, which is called a “kernel launch”.

Example 2: add 2 numbers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}  
int main(void) {  
    int ha=1,hb=2,hc;  
    add<<<1,1>>>(&ha, &hb, &hc);  
    printf("c=%d\n",hc);  
    return 0;  
}
```



- This does not work!!
- `int ha, hb, hc` are in the host memory (DRAM), which cannot be used by device (GPU).
- We need to allocate variables in “device memory”.

The correct main ()

```
int main(void) {  
    int a=1, b=2, c; // host copies of a, b, c  
    int *d_a, *d_b, *d_c; // device copies of a, b, c  
    // Allocate space for device copies of a, b, c  
    cudaMalloc((void **)&d_a, sizeof(int));  
    cudaMalloc((void **)&d_b, sizeof(int));  
    cudaMalloc((void **)&d_c, sizeof(int));  
    // Copy inputs to device  
    cudaMemcpy(d_a, &a, sizeof(int), cudaMemcpyHostToDevice);  
    cudaMemcpy(d_b, &b, sizeof(int), 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;  
}
```

Example 3: add 2 vectors

■ Let's first look at the sequential code!

```
// function definition
void VecAdd(int N, float* A, float* B, float* C)
{
    for(int i = 0; i<N; i++)
        C[i] = A[i] + B[i];
}

int main()
{ ...
    VecAdd (N, Ah, Bh, Ch);
    ...
}
```

Parallel CUDA code

- Use `blockIdx.x` as the index of the arrays
 - Each thread processes 1 addition, for the elements indexed at `blockIdx.x`.

```
// 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>>>(Ah, Bh, Ch); ...
}
```

Alternative implementation

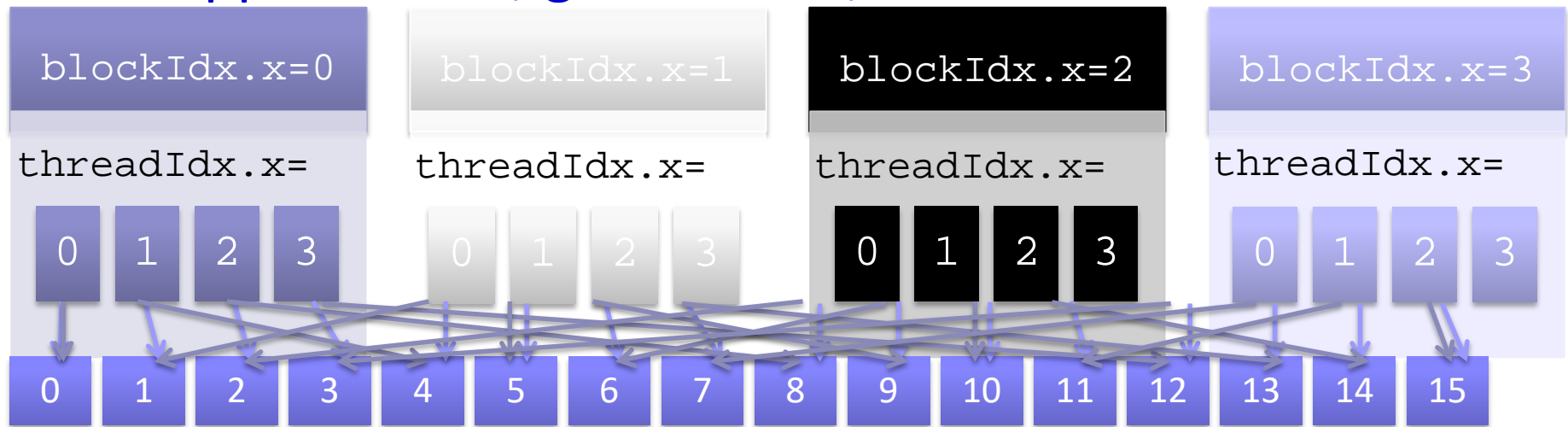
- Using parallel thread instead

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}  
int main(void) {  
    int a[N], b[N], c[N];  
    int *d_a, *d_b, *d_c;  
    ...  
    add<<< N, 1 >>>(d_a, d_b, d_c);  
    ...  
}
```

- N blocks and each block has 1 thread.
- Which one is better?
 - Threads in the same block can communicate, synchronize with others, but the number of threads per block is limited.

3rd implementation

- Using multiple threads and multiple blocks
- Suppose $N=16$, grid size = 4, and block size = 4



- How to index 16 elements of an array?
 - Method 1: $\text{index} = \text{blockIdx.x} * 4 + \text{threadIdx.x}$
 - Method 2: $\text{index} = \text{threadIdx.x} * 4 + \text{blockIdx.x}$
- Which one is better?

The general case

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

```
__global__ void add(int *a, int *b, int *c) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    c[index] = a[index] + b[index];  
}  
  
int main(void) {  
    int a[N], b[N], c[N];  
    int *d_a, *d_b, *d_c;  
    ...  
    add<<< N/BS, BS >>>>(d_a, d_b, d_c);  
    ...  
}
```

What if N is not a multiple of BS?

- BS is block size (number of threads per block)

A even more general case

```
__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];  
}  
int main(void) {  
    int a[N], b[N], c[N];  
    int *d_a, *d_b, *d_c;  
    ...  
    add<<< (N+BS-1)/BS, BS>>>(d_a, d_b, d_c, N);  
    ...  
}
```

- The kernel function can have branches, but with a price to pay...

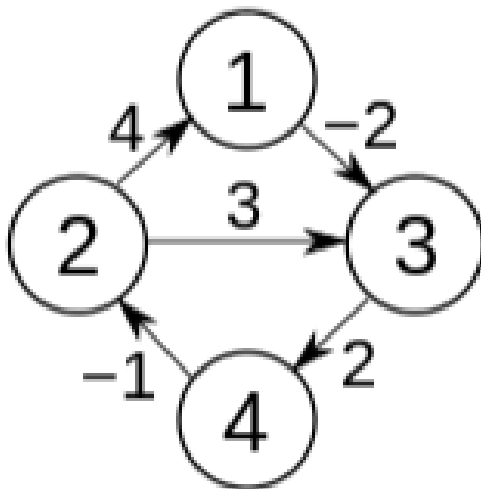
Outline

- CUDA Basic
- Example Code Study
 - Hello world
 - All pairs shortest path
- Time Measurement & Debug API
- Multi-GPUs
- Dynamic Parallelism

APSP

- Given a weighted directed graph $G(V, E, W)$, where $|V| = n$, $|W| = m$, and $W > 0$, find the shortest path of all pairs of vertices (v_i, v_j) .

- Example:



0	INF	-2	INF
4	0	3	INF
INF	INF	0	2
INF	-1	INF	0

Initial
weight

0	-1	-2	0
4	0	2	4
5	1	0	2
3	-1	1	0

Final
result

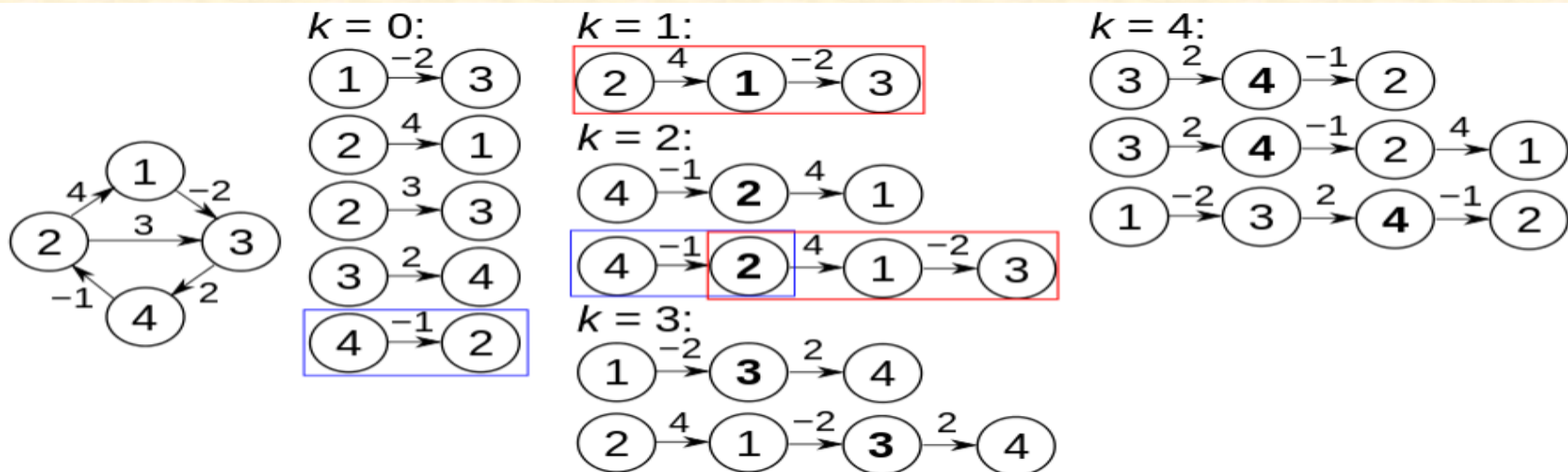
Floyd-Warshall (Sequential code)

Floyd-Warshall(G, W)

```

{
   $n \leftarrow |V|$ 
   $D^{(0)} \leftarrow W$ 
  for  $k = 1$  to  $n$  do
    for  $i = 1$  to  $n$  do
      for  $j = 1$  to  $n$  do
        if  $D^{(k-1)}[i, j] > D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$ 
        then  $D^{(k)}[i, j] \leftarrow D^{(k-1)}[i, k] + D^{(k-1)}[k, j]$ 
        else  $D^{(k)}[i, j] \leftarrow D^{(k-1)}[i, j]$ 
  return  $D^{(n)}$ 
}
```

How to parallelize it?



Implementation I

- 1 block and n threads.
- Thread i updates the SP for vertex i .

```
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    for (int j = 0; j < n; j++)  
        if (D[i][j] > D[i][k] + D[k][j])  
            D[i][j] = D[i][k] + D[k][j];  
}  
  
int main() { ...  
    for (int k = 0; k < n; k++)  
        FW_APSP<<<1, n>>>(k, D);  
}
```

Simple! But can it be faster ?

Implementation 2

■ Each thread updates one pair of vertices

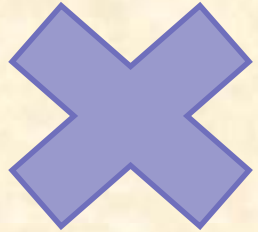
➤ Increase parallelism from n to n^2

```
__global__ void FW_APSP(int k, int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    if (D[i][j] > D[i][k] + D[k][j])  
        D[i][j] = D[i][k] + D[k][j];  
}  
  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    for (int k = 0; k < n; k++)  
        FW_APSP<<<1, threadsPerBlock >>>(k, D);  
}
```

■ How about the for-loop of k ?

Implementation 3

```
__global__ void FW_APSP(int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    for (int k = 0; k < n, k++)  
        if (D[i][j] > D[i][k] + D[k][j])  
            D[i][j] = D[i][k] + D[k][j];  
}  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    FW_APSP<<<1, threadsPerBlock >>>(D);  
}
```



- It is a synchronous computation
 - There are data dependency on k...

Add __syncthreads ()

```
__global__ void FW_APSP(int D[n][n]) {  
    int i = threadIdx.x;  
    int j = threadIdx.y;  
    for (int k = 0; k < n, k++){  
        if (D[i][j] > D[i][k] + D[k][j])  
            D[i][j] = D[i][k] + D[k][j];  
        __syncthreads();  
    }  
}  
  
int main() { ...  
    dim3 threadsPerBlock(n, n);  
    FW_APSP<<<1, threadsPerBlock>>>(D);  
}
```




Outline

- CUDA Basic
- Example Code Study
- Time Measurement & Debug API
- Multi-GPUs
- Dynamic Parallelism

Execution time: in host

- CUDA provides functions to measure the execution time between events.

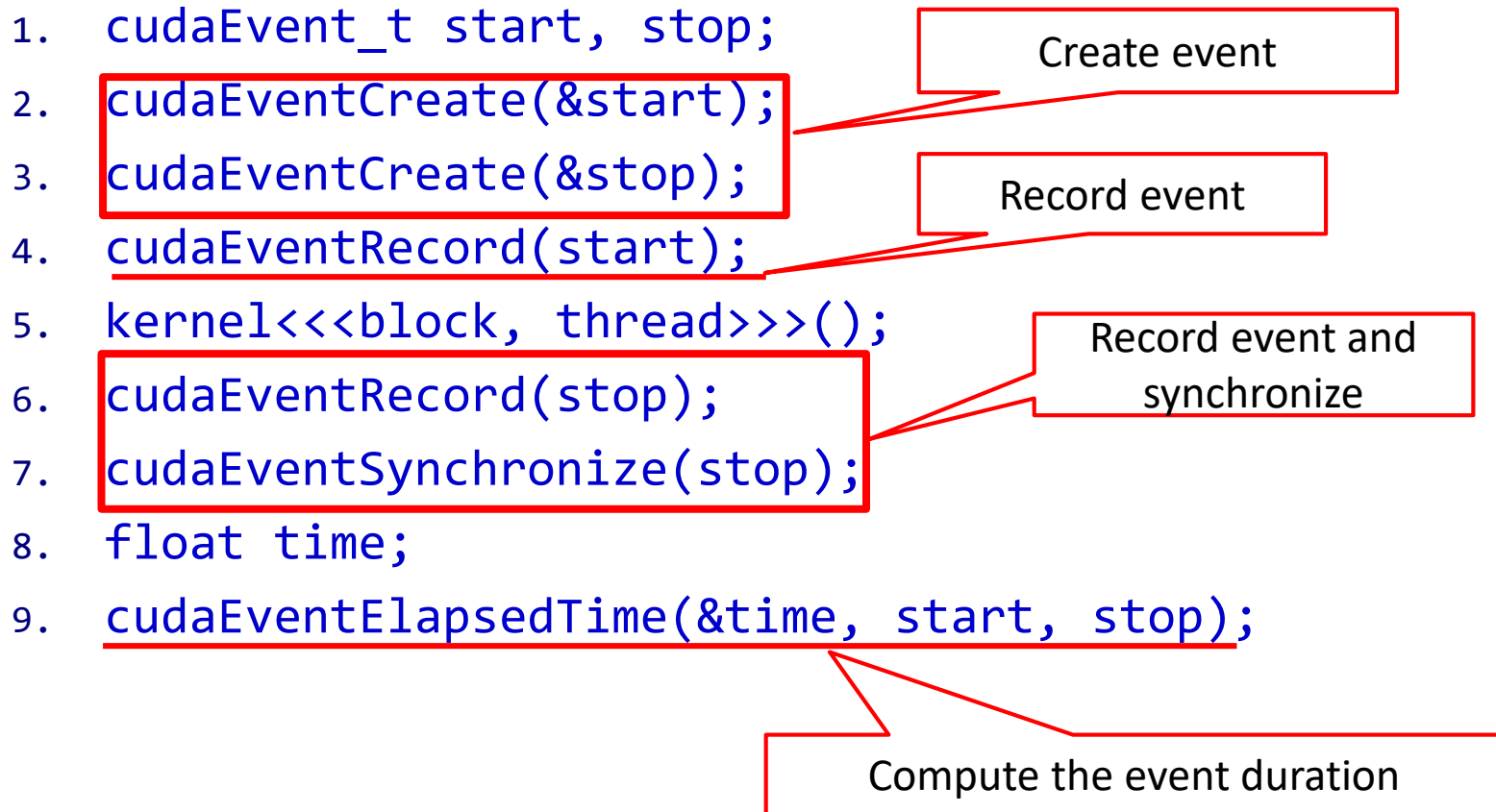
```
cudaError_t cudaEventElapsedTime(  
    float* ms,  
    cudaEvent_t start,  
    cudaEvent_t end)
```

- ms: time between start and end in ms
 - start: starting event
 - end: ending event
- The time unit is milliseconds, whose resolution is 0.5 microseconds

CUDA event

- Data type : `cudaEvent_t`
- `cudaError_t cudaEventCreate(cudaEvent_t* event)`
 - Create CUDA event
- `cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0)`
 - Record CUDA event
 - If stream is non-zero, the event is recorded after all preceding operations in the stream have been completed
 - Since operation is **asynchronous**, `cudaEventQuery()` and/or `cudaEventSynchronize()` must be used to determine when the event has actually been recorded
- `cudaError_t cudaEventSynchronize(cudaEvent_t event)`
 - Wait until the completion of all device work preceding the most recent call to **`cudaEventRecord()`**

Example



Reporting errors

- All CUDA calls return an error code `cudaError_t`
 - Error in the API call itself OR **Error in an earlier asynchronous operation** (e.g. kernel)
- Get the error code for the **last** error:
 - `cudaError_t cudaGetLastError(void)`
- Get a string to describe the error:
 - `char *cudaGetErrorString(cudaError_t)`



Outline

- CUDA Basic
- Example Code Study
- Time Measurement & Debug API
- **Multi-GPUs**
- Dynamic Parallelism

Using CUDA with pthread

■ Launch the kernel function for a GPU in a thread

```
void* GPUthread(void* arg){
    struct HYBctx* ctx = (struct HYBctx*)arg;
    int *dA, A[32]=...;
    cudaMalloc((void**)&dA, sizeof(int)*32);
    cudaMemcpy(dA, A, sizeof(int)*32,
               cudaMemcpyHostToDevice);
    kernel<<<1, 32>>>(dA);
    cudaMemcpy(A, dA, sizeof(int)*32,
               cudaMemcpyDeviceToHost);
    cudaFree(dA);
    cudaDown(ctx);
    return NULL;
}
```

Using CUDA with OpenMP

- Put CUDA functions inside the parallel region
- General setting:
 - The number of CPU threads is the same as the number of CUDA devices. Each CPU thread controls a different device, processing its portion of the data.
- It's possible to use more CPU threads than there are CUDA devices.
 - Several CPU threads will be allocating resources and launching kernel on the same device, which will slow down the performance.

Example: cudaOMP.cu

```
...
cudaGetDeviceCount(&num_gpus);
...
omp_set_num_threads(num_gpus);
// create as many CPU threads as there are CUDA devices
#pragma omp parallel
{
    unsigned int cpu_thread_id = omp_get_thread_num();
    unsigned int num_cpu_threads = omp_get_num_threads();
    CUDA_SAFE_CALL(cudaSetDevice(cpu_thread_id));
    int gpu_id = -1;
    CUDA_SAFE_CALL(cudaGetDevice(&gpu_id));
    printf("CPU thread %d (of %d) uses CUDA device %d\n",
          cpu_thread_id, num_cpu_threads, gpu_id);
    ...
}
```

Using CUDA with MPI

```
int main(int argc, char* argv){
    int rank, size;
    int A[32];
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);
    for(i = 0; i < 32; i++) A[i] = rank+1;
    launch(A); // a call to launch CUDA kernel
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Example: **launch(A)**

```
extern "C"
void launch(int *A){
    int *dA;
    cudaMalloc((void**)&dA, sizeof(int)*32);
    cudaMemcpy(dA, A, sizeof(int)*32,
               cudaMemcpyHostToDevice);
    kernel<<<1, 32>>>(dA);
    cudaMemcpy(A, dA, sizeof(int)*32,
               cudaMemcpyDeviceToHost);
    cudaFree(dA);
}
```

Compilation and execution

■ Compilation

- `nvcc -c kernel.cu`
- `mpicc -o mpicuda mpi.c kernel.o`
`-L /usr/local/cuda/lib -lcudart`
`-I /usr/local/cuda/include`

■ Execution

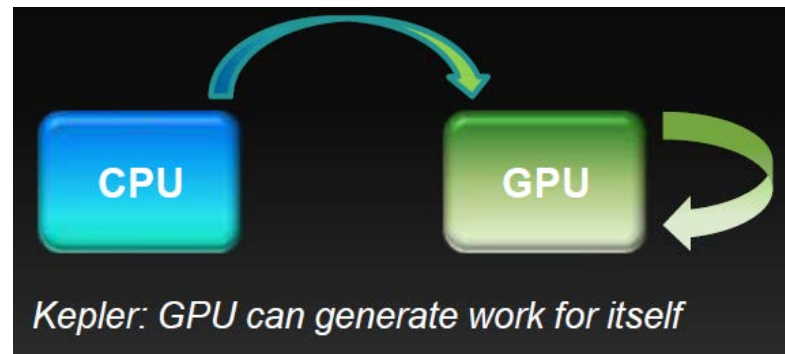
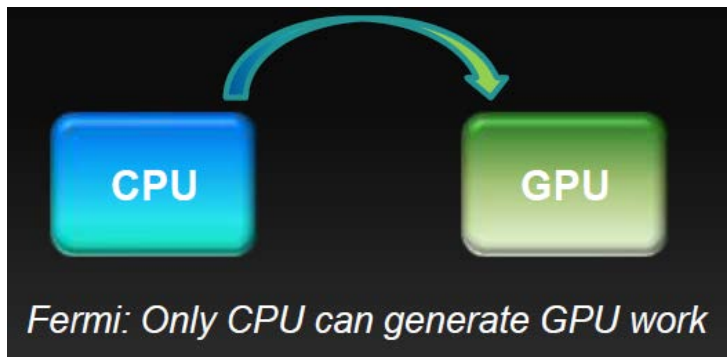
- `mpirun -l -np 4 ./mpicuda`

Outline

- CUDA Basic
- Example Code Study
 - Hello world
 - All pairs shortest path
- Time Measurement & Debug API
- Multi-GPUs
- **Dynamic Parallelism**

Dynamic parallelism

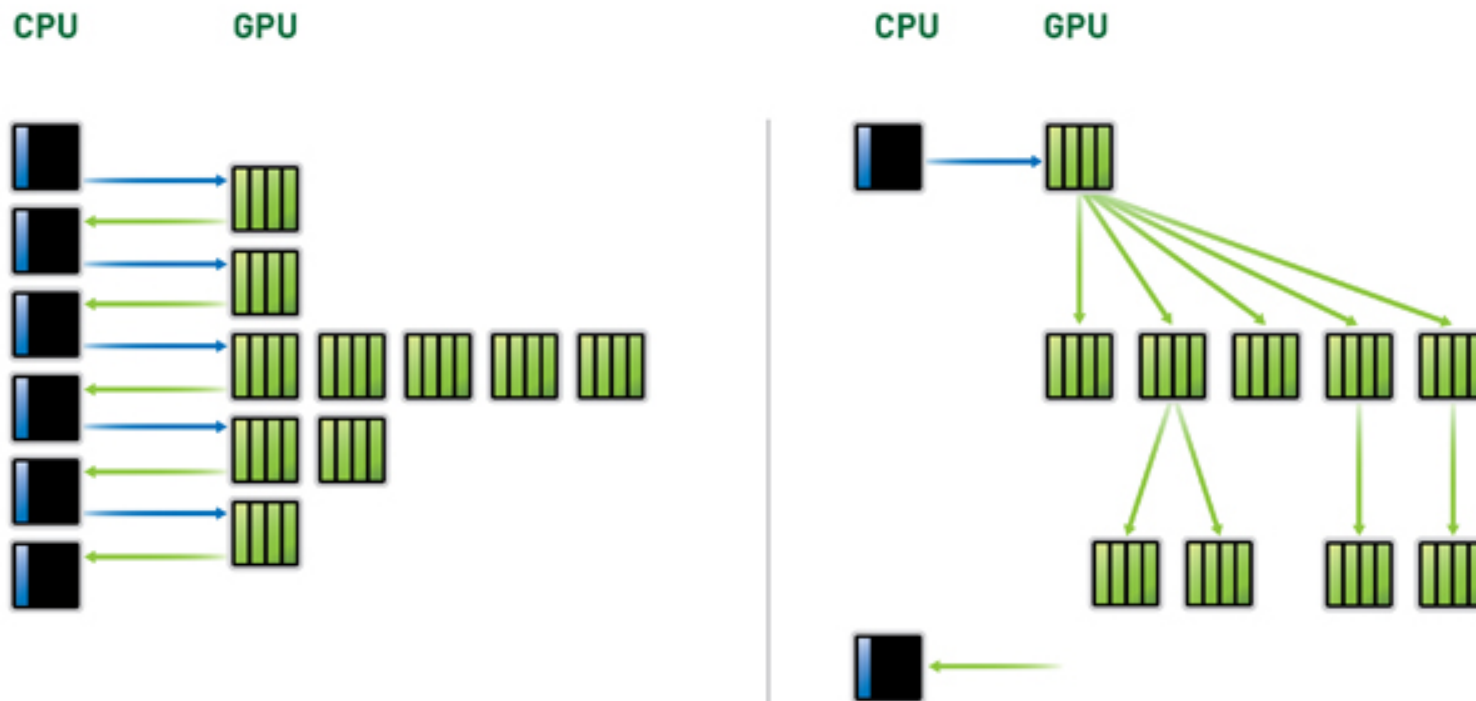
- The ability to launch new **grids** from the **GPU**
 - Dynamically
 - Simultaneously
 - Independently
- Supported from **CUDA5.0** on devices of **Compute Capability 3.5** or higher



What does it mean?

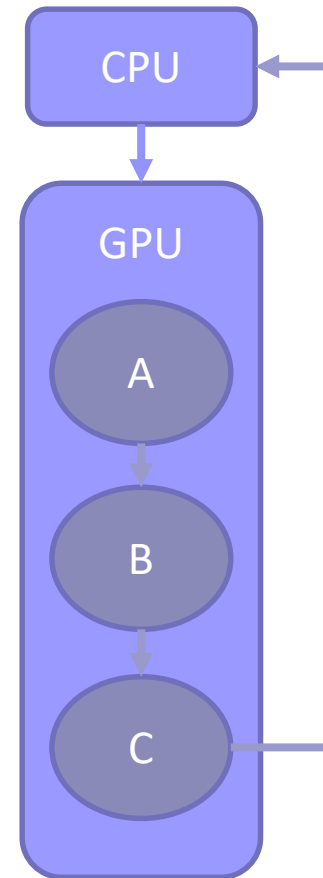
- Reduce the number of kernel launches

DYNAMIC PARALLELISM



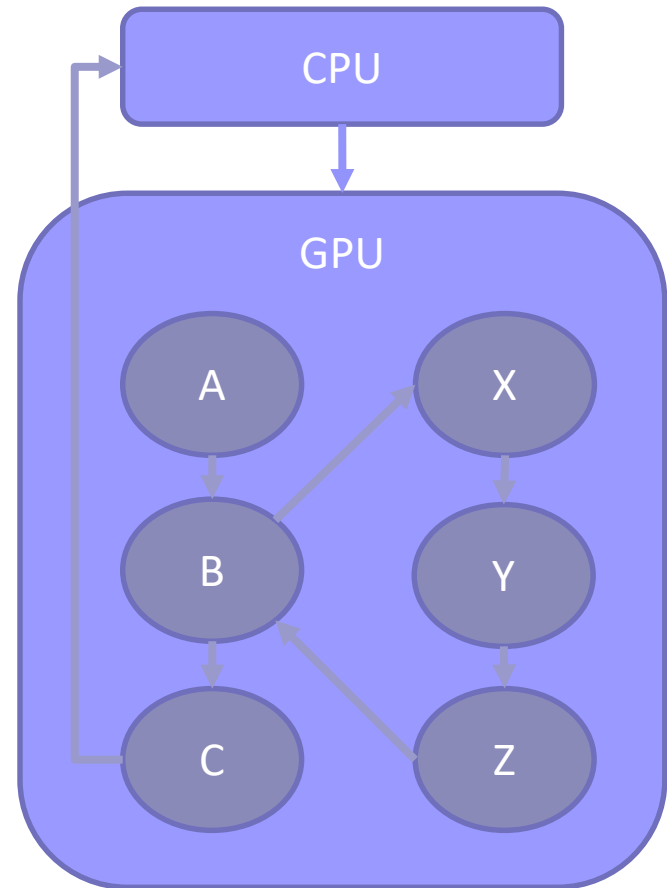
Dependency in CUDA

```
void main() {  
    float *data;  
    do_stuff(data);  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
    cudaDeviceSynchronize();  
    do_more_stuff(data);  
}
```



Nested dependency

```
void main() {  
    float *data;  
    do_stuff(data);  
    A <<< ... >>> (data);  
    B <<< ... >>> (data);  
    C <<< ... >>> (data);  
    cudaDeviceSynchronize();  
    do_more_stuff(data);  
}  
  
__global__ void B(float *data){  
    do_stuff(data);  
    X <<< ... >>> (data);  
    Y <<< ... >>> (data);  
    Z <<< ... >>> (data);  
    cudaDeviceSynchronize();  
    do_more_stuff(data);  
}
```

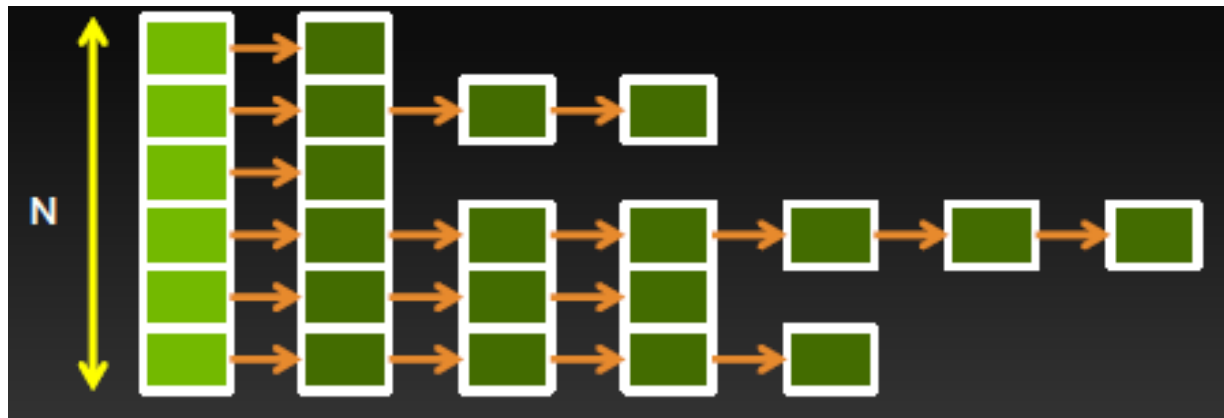


What is DP good for?

- Dynamic block size and grid size
- Dynamic work generation
- Nested parallelism
- Library calls
- Parallel recursion

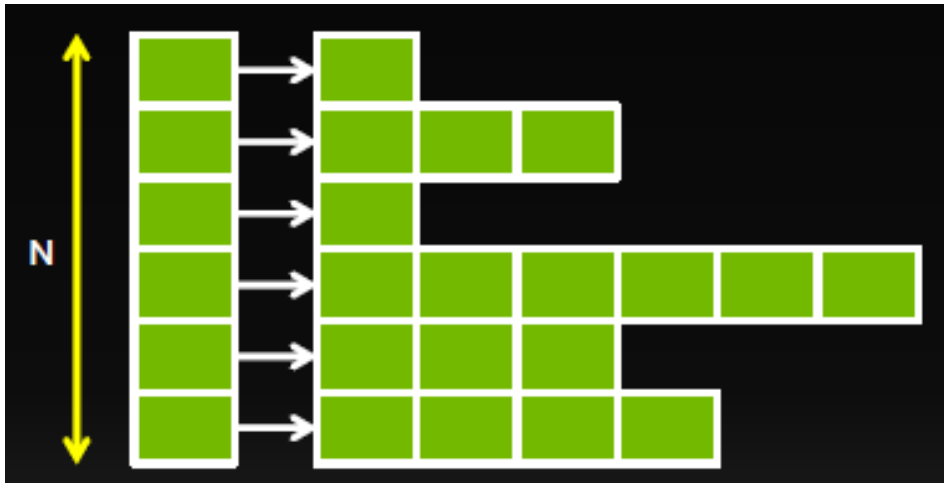
1. Dynamic block size and grid size

```
for i = 1 to N
  for j = 1 to x[i]
    convolution(i, j)
  next j
next i
```

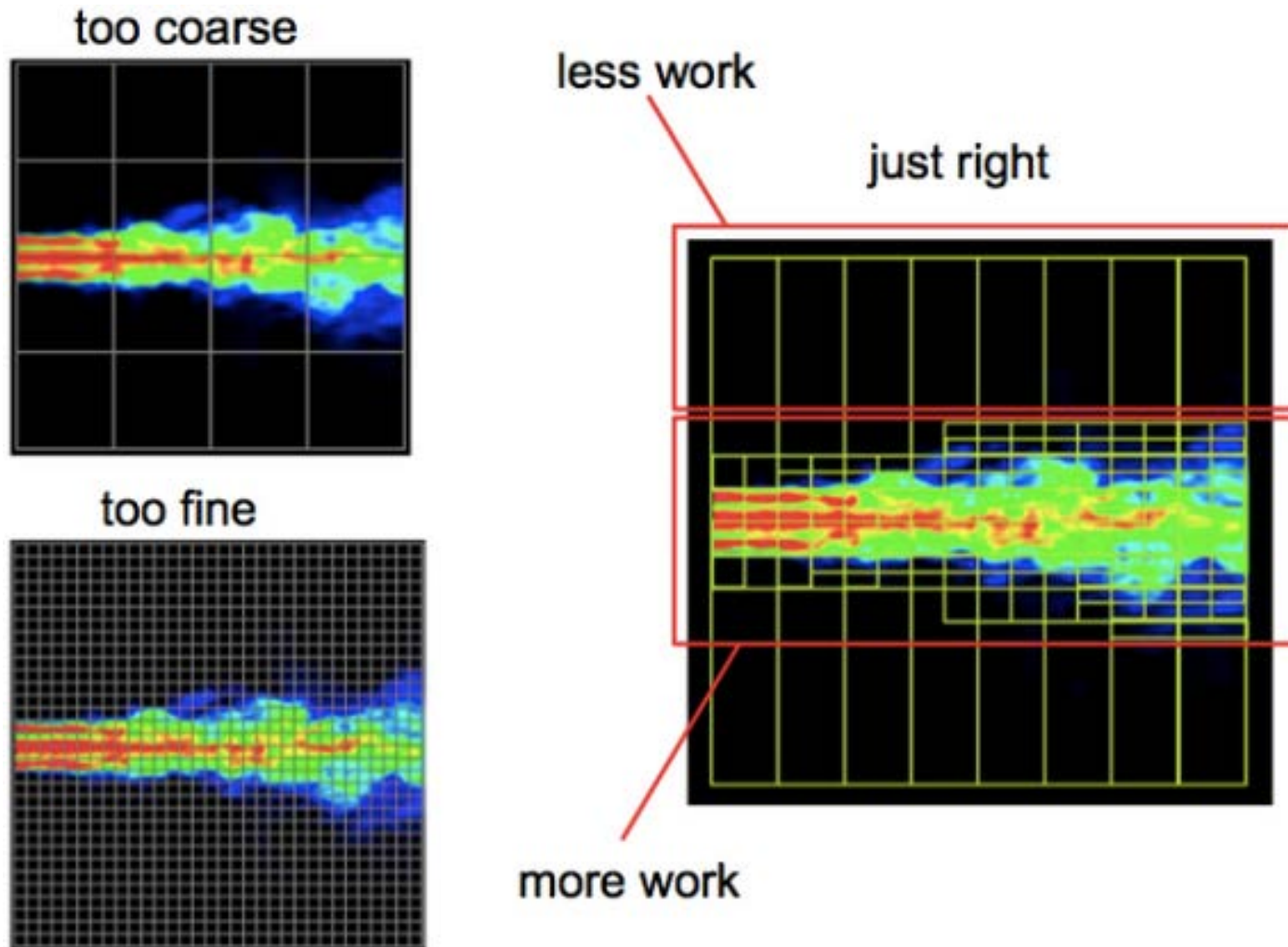


1. Dynamic block size with DP

```
__global__ void convolution(int x[]){  
    for j = 1 to x[blockIdx]  
        kernel<<< ... >>>(blockIdx, j)  
    }  
    ...  
    convolution<<< N, 1 >>>(x);  
}
```



2. Dynamic work generation



3. Nested Parallelism

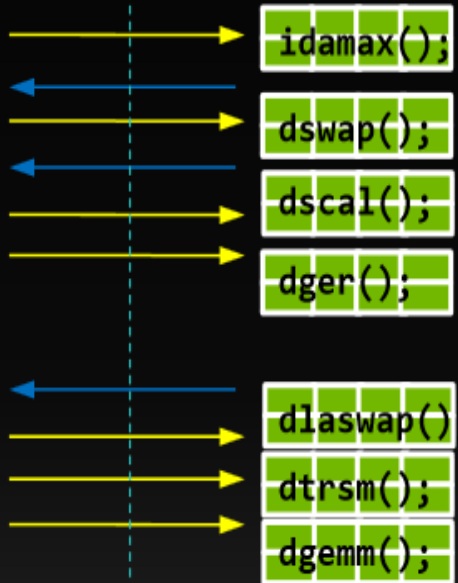
LU decomposition (Fermi)

```

dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>
      memcpy
      dswap<<<>>
      memcpy
      dscal<<<>>
      dger<<<>>
    next i

    memcpy
    dlaswap<<<>>
    dtrsm<<<>>
    dgemm<<<>>
  next j
}
    
```

CPU Code



GPU Code

LU decomposition (Kepler)

```

dgetrf(N, N) {
  dgetrf<<<>>
}
    
```

CPU Code

```

dgetrf(N, N) {
  for j=1 to N
    for i=1 to 64
      idamax<<<>>
      dswap<<<>>
      dscal<<<>>
      dger<<<>>
    next i
    dlaswap<<<>>
    dtrsm<<<>>
    dgemm<<<>>
  next j
}
    
```

GPU Code

```

synchronize();
}
    
```

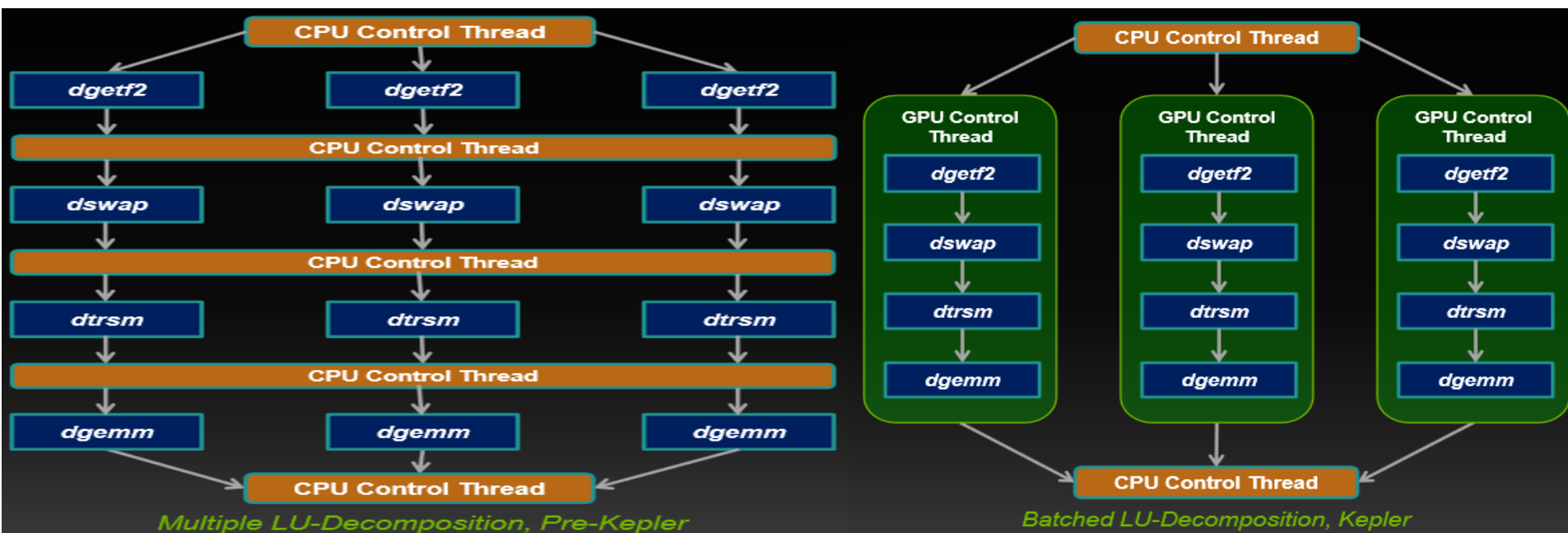
LU Decomposition

■ CPU controlled batching

- Limited by single point control
- Can run at most 10s of threads
- CPU is fully consumed with controlling launches

■ Batching via DP

- Move top loop to GPU
- Run thousands of independent tasks
- Release CPU for other work



4. Library call

```
__global__ void libraryCall(float *a, float *b, float *c){
    // All threads generate data
    createData(a, b);
    __syncthreads();
    // Only one thread calls library
    if(threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }
    // All threads wait for dtrsm
    __syncthreads();
    // Now continue
    consumeData(c);
}
```


5. Parallel Recursion

■ Quick sort

```
__global__ void qsort(int *data, int l, int r) {  
    int pivot = data[0];  
    int *lptr = data+l, *rptr = data+r;  
  
    // Partition data around pivot value  
    partition(data, l, r, lptr, rptr, pivot);  
  
    // Launch next stage recursively  
    if(l < (rptr-data))  
        qsort<<< ... >>>(data, l, rptr-data);  
    if(r > (lptr-data))  
        qsort<<< ... >>>(data, lptr-data, r);  
}
```

Reference

- **NVIDIA CUDA Library Documentation**

http://developer.download.nvidia.com/compute/cuda/4_1/rel/toolkit/docs/online/index.html

- Heterogeneous computing course slides from Prof. Che-Rung Lee