北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Chapter 7
GPU Parallel Programming
CUDA

软件学院　邵兵

2022年4月19日

Contents

北京航空航天大學

COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

What is CUDA?

CUDA: Compute Unified Device Architecture

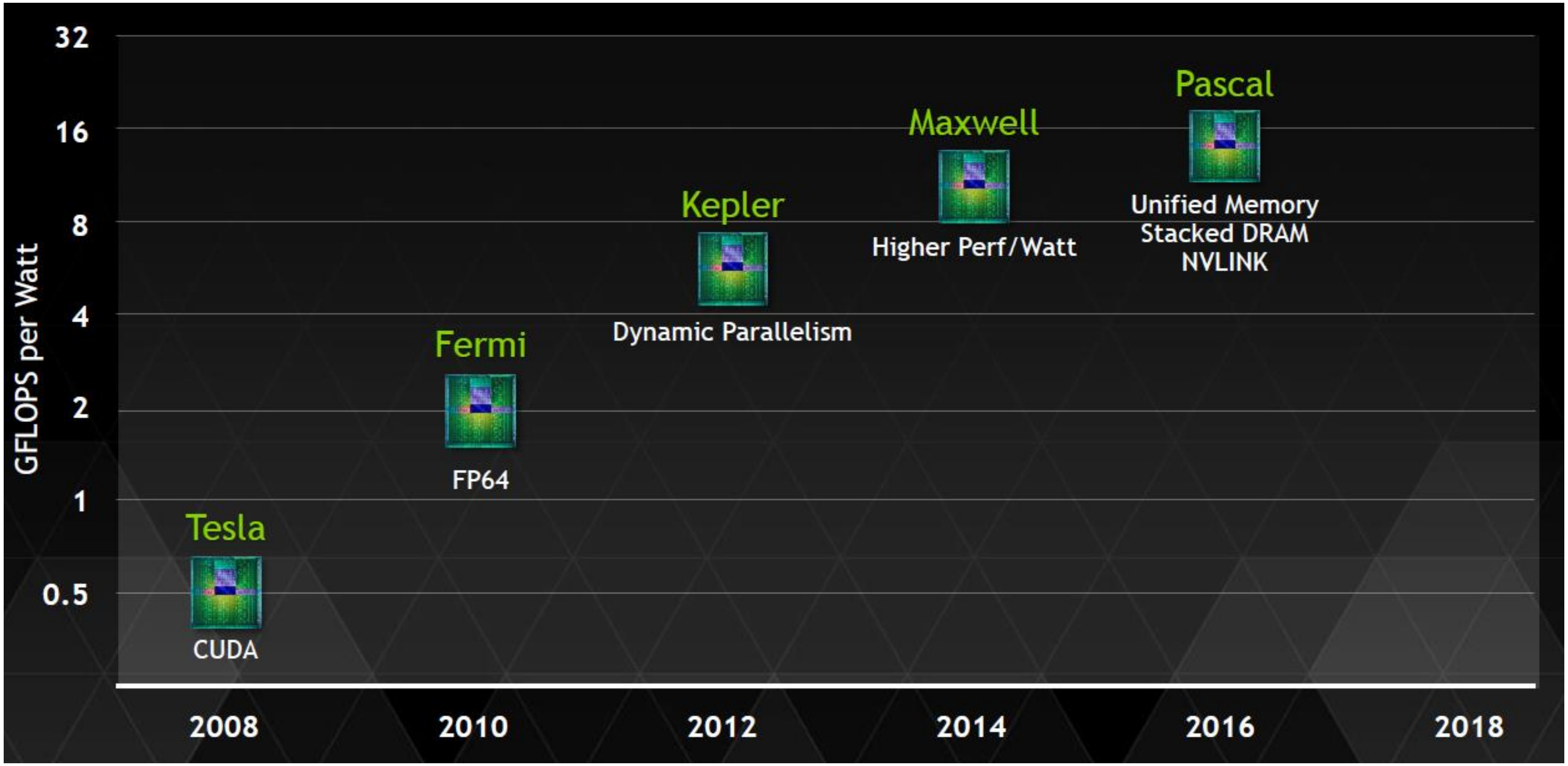
- CUDA is a compiler and toolkit for programming NVIDIA GPUs
- Enable heterogeneous computing and horsepower of GPUs
- CUDA API extends the C/C++ programming language
- Express SIMD parallelism
- Give a high level abstraction from hardware

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

CUDA Version

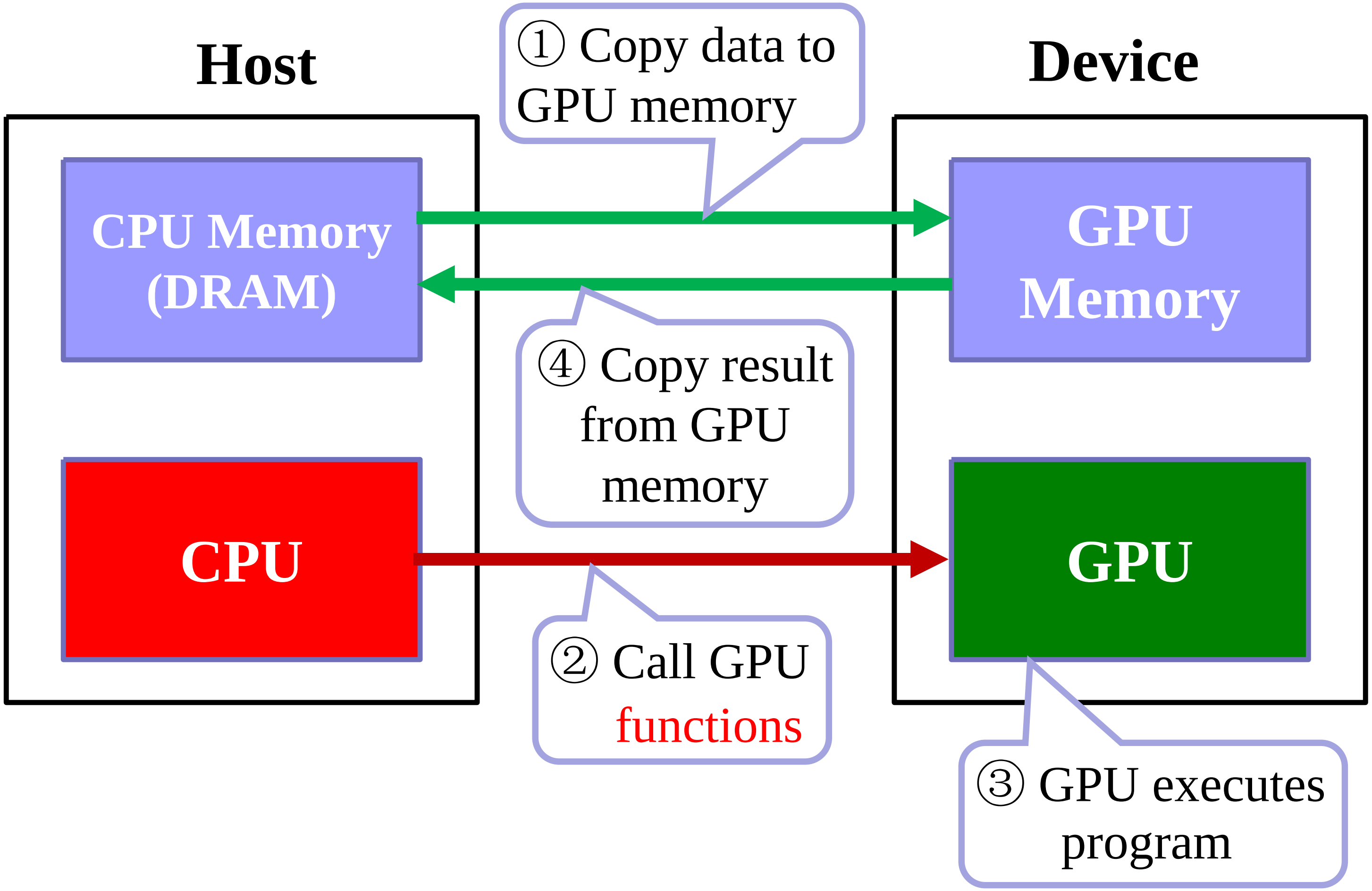
CUDA SDK Version	Compute Capability	Architecture
6.5	1.X	Tesla
7.5	2.0-5.x	Fermi, Kepler, Maxwell
8.0	2.0-6.x	Fermi, Kepler, Maxwell, Pascal
9.0	3.0-7.x	Kepler, Maxwell, Pascal, Volta
10.2	6.1x	... Turing

https://www.cnblogs.com/timlly/p/11471507.html



北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

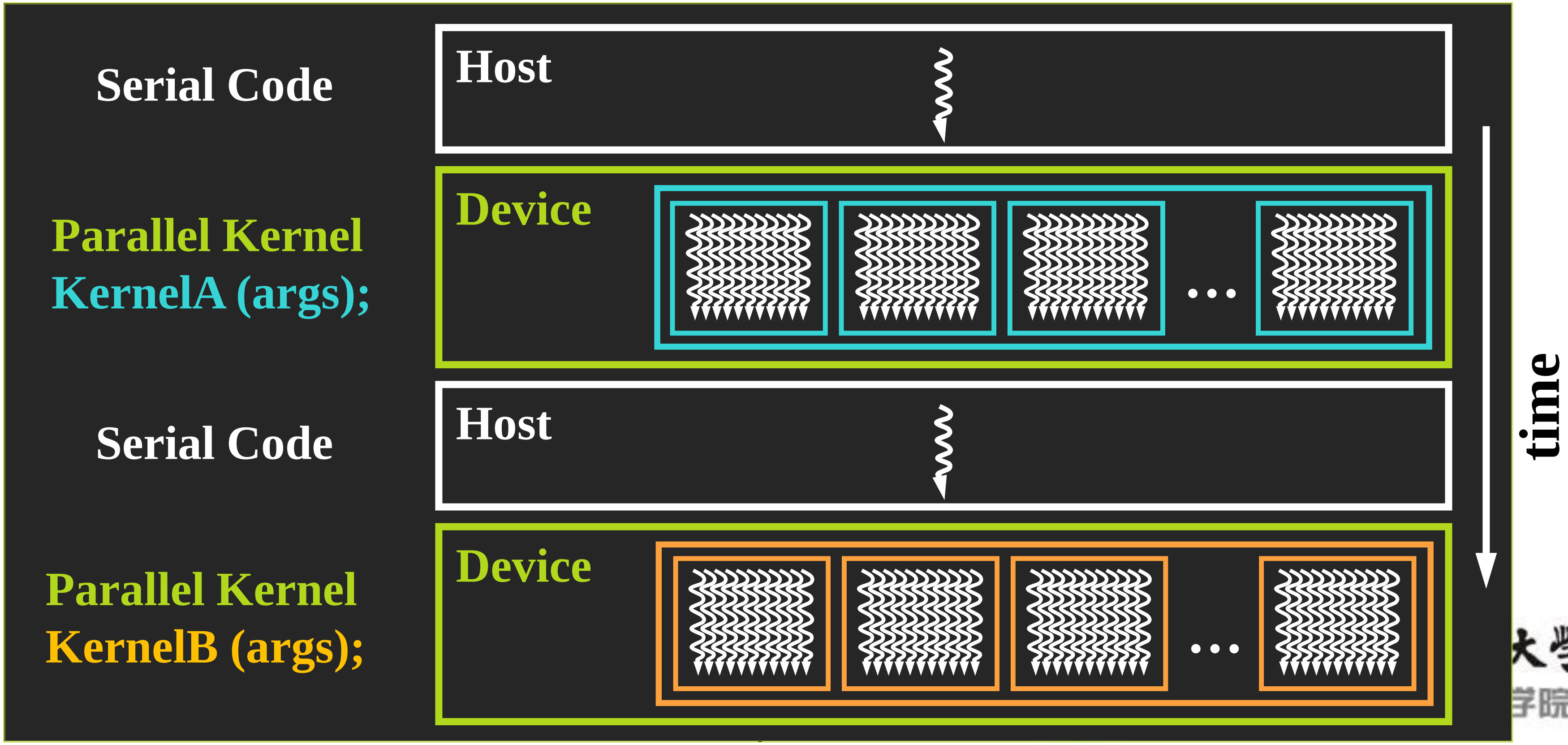
CUDA Program Flow



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

```
#include <cuda_runtime.h>

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

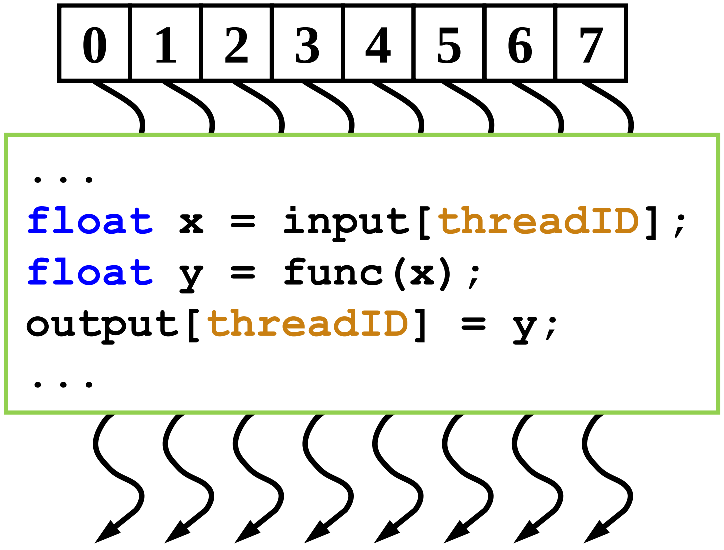
CPU code (serial or parallel if pthread/ OpenMP/ MPI is used.)

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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Kernel = Many Concurrent Threads

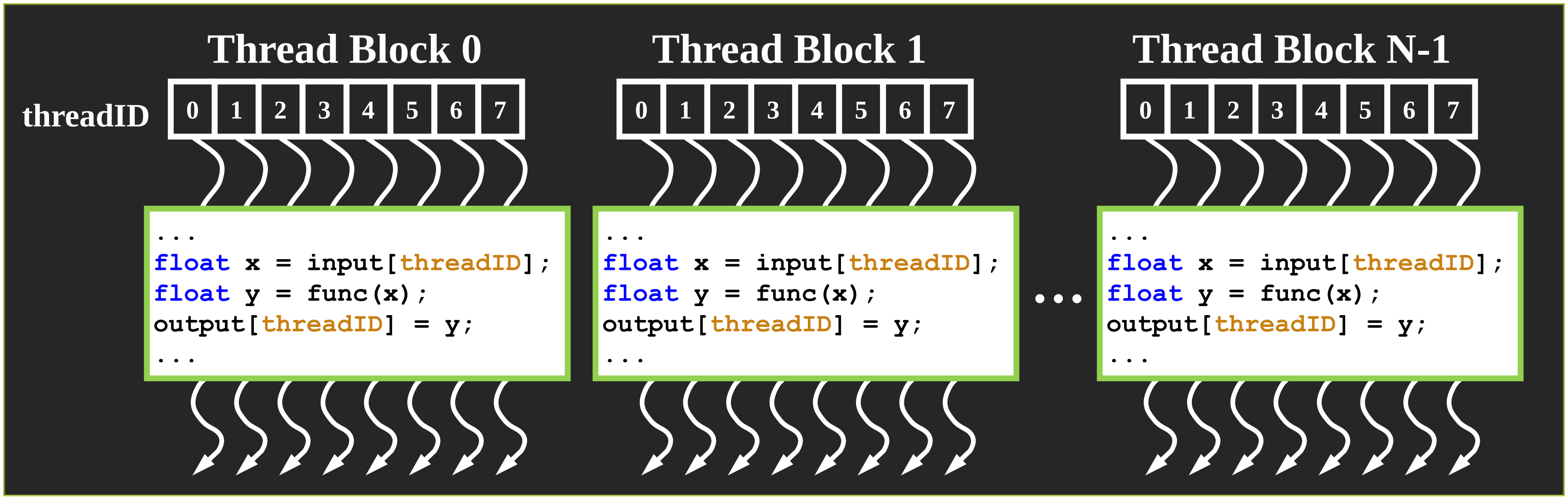
- **One kernel is executed at a time on the device**
- **Many thread execute each kernel**
  - Each thread executes the same code
  - … on the different data based on its threadID
- **CUDA thread might be**
  - Physical threads
    - As on NVIDIA GPUs
    - GPU thread creation and context switching are 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 = gird of thread blocks

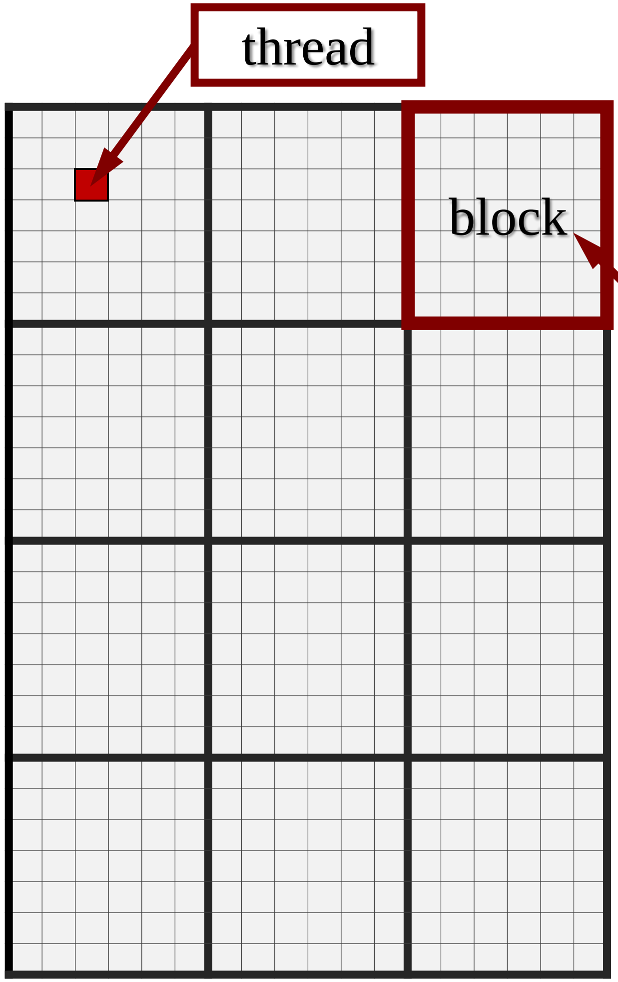


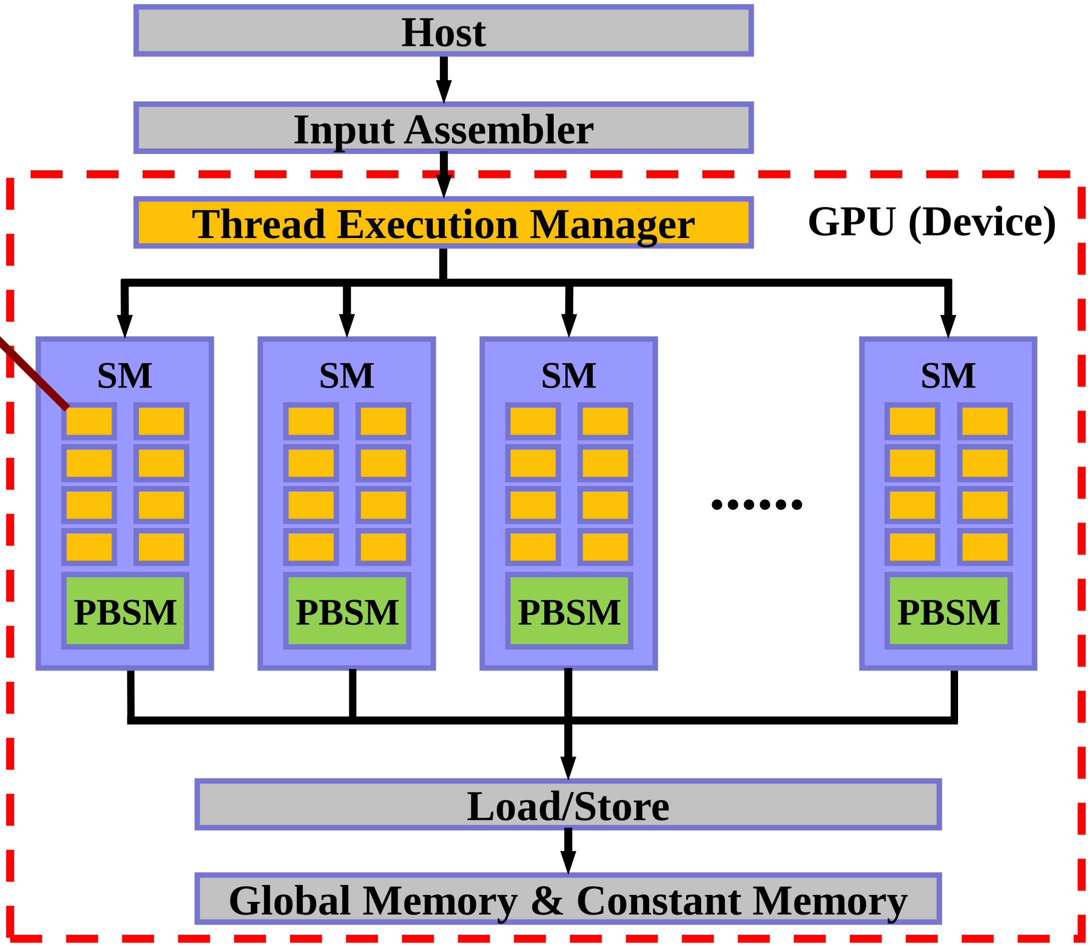
- **By definition, threads in the same block may synchronized with barriers, but not between blocks.**

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

Software Mapping

- **Software:** grid → blocks → threads
- **Hardware:** GPU(device) → SM(Stream Multiprocessor) → core





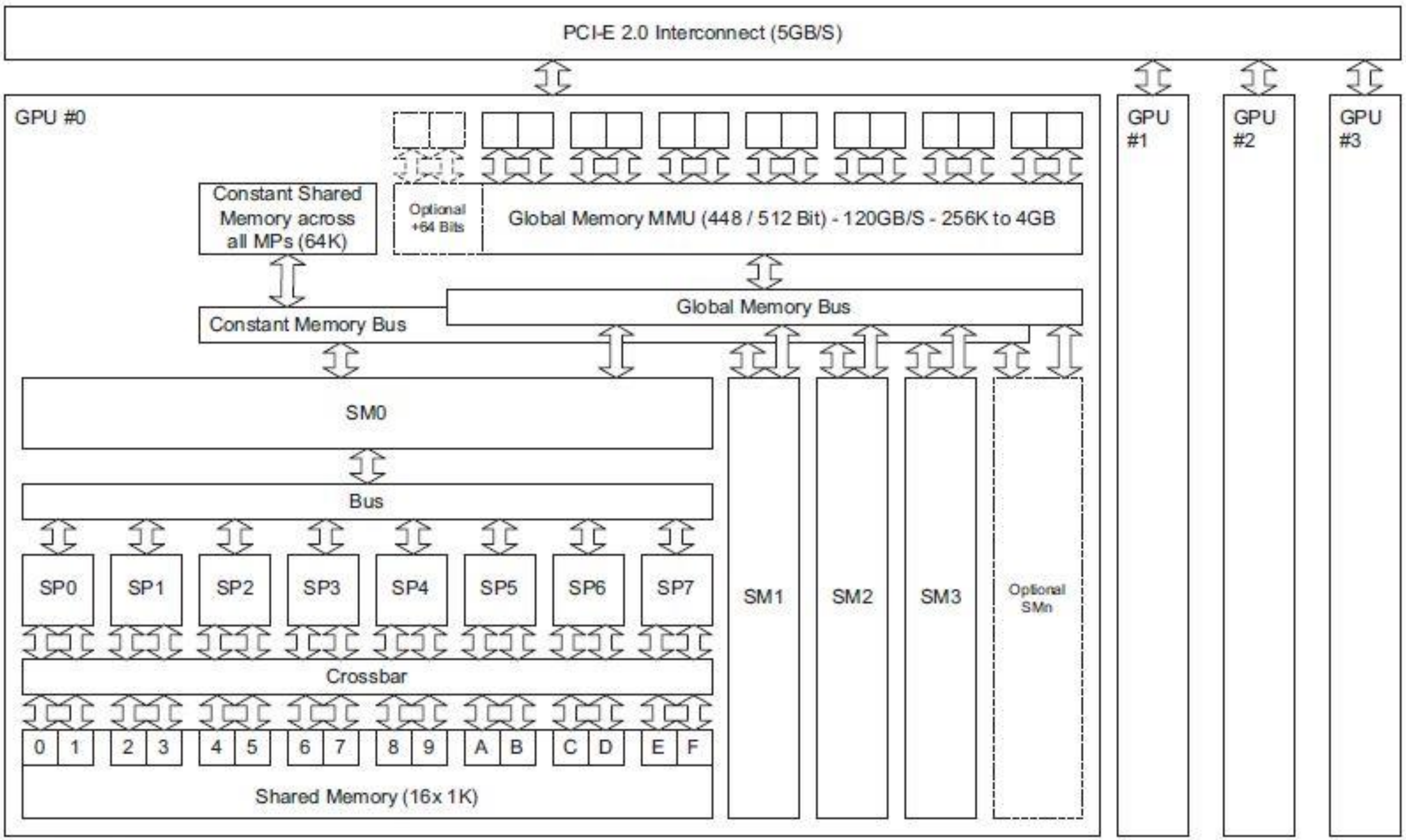
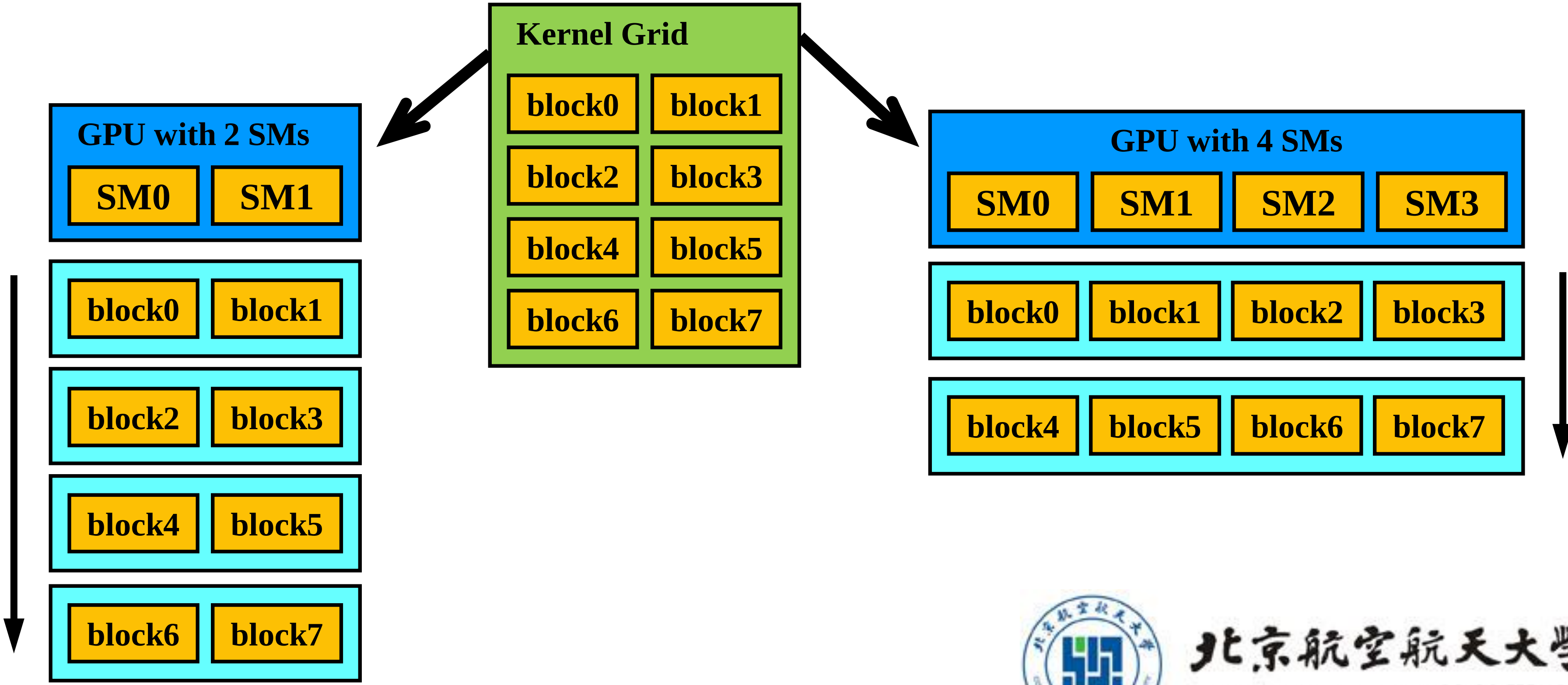

PCI-E 2.0 Interconnect (5GB/S)
GPU #0
GPU #1
GPU #2
GPU #3
Constant Shared Memory across all MPs (64K)
Optional +64 Bits
Global Memory MMU (448 / 512 Bit) - 120GB/S - 256K to 4GB
Global Memory Bus
Constant Memory Bus
SM0
Bus
SP0
SP1
SP2
SP3
SP4
SP5
SP6
SP7
Crossbar
0 1 2 3 4 5 6 7 8 9 A B C D E F
Shared Memory (16x 1K)
SM1
SM2
SM3
Optional SMn

FIGURE 3.5

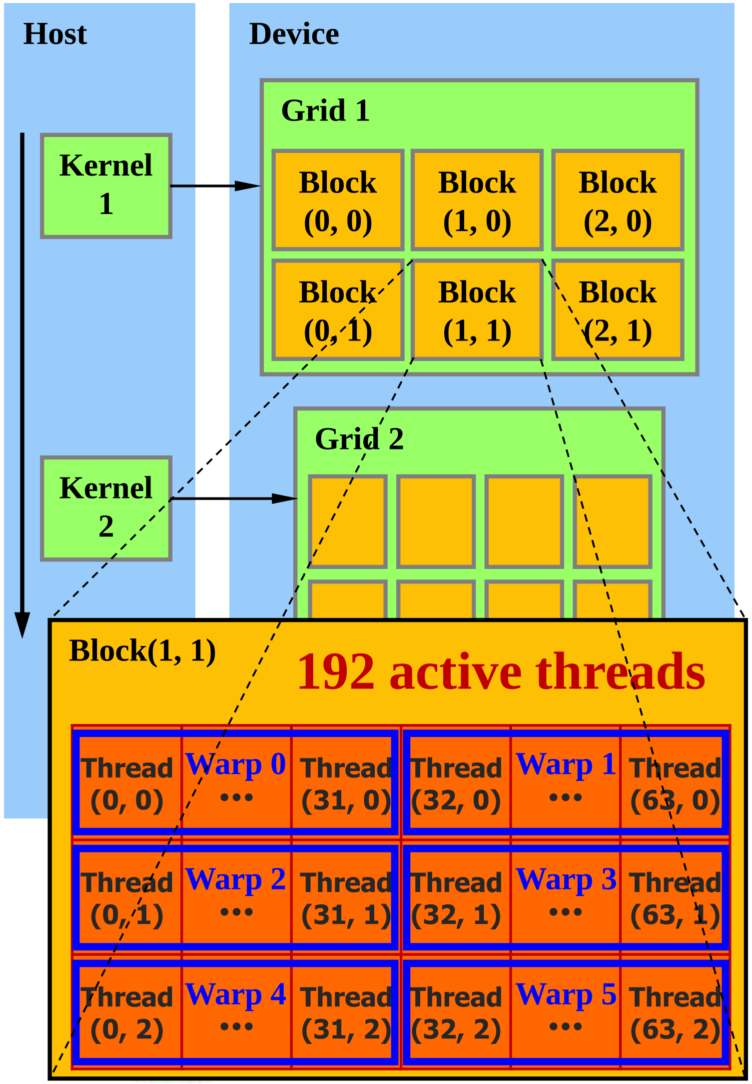
Block Level Scheduling

- **Blocks are independent to each other to give scalability**
  - A kernel scales across any number of parallel cores by scheduling blocks to SMs

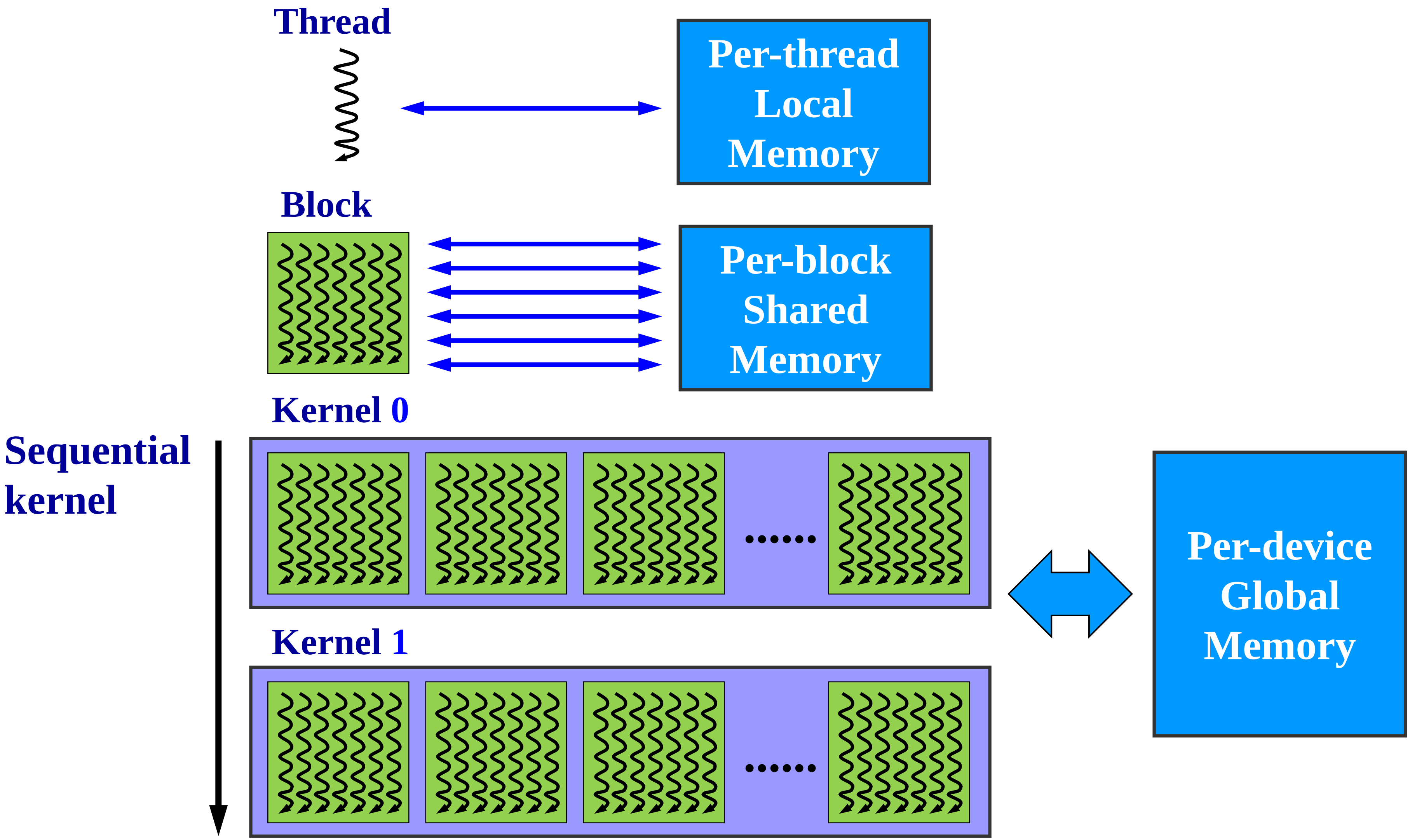


Thread Level Scheduling - Warp

- **Inside the SM, threads are launched in groups of 32, called warps**
  - Warps share the control part (warp scheduler)
  - Threads in a warp will be executing the same instruction (SIMD)
- **In other words …**
  - Threads in a wrap execute **physically** in parallel
  - Warps and blocks execute **logically** in parallel

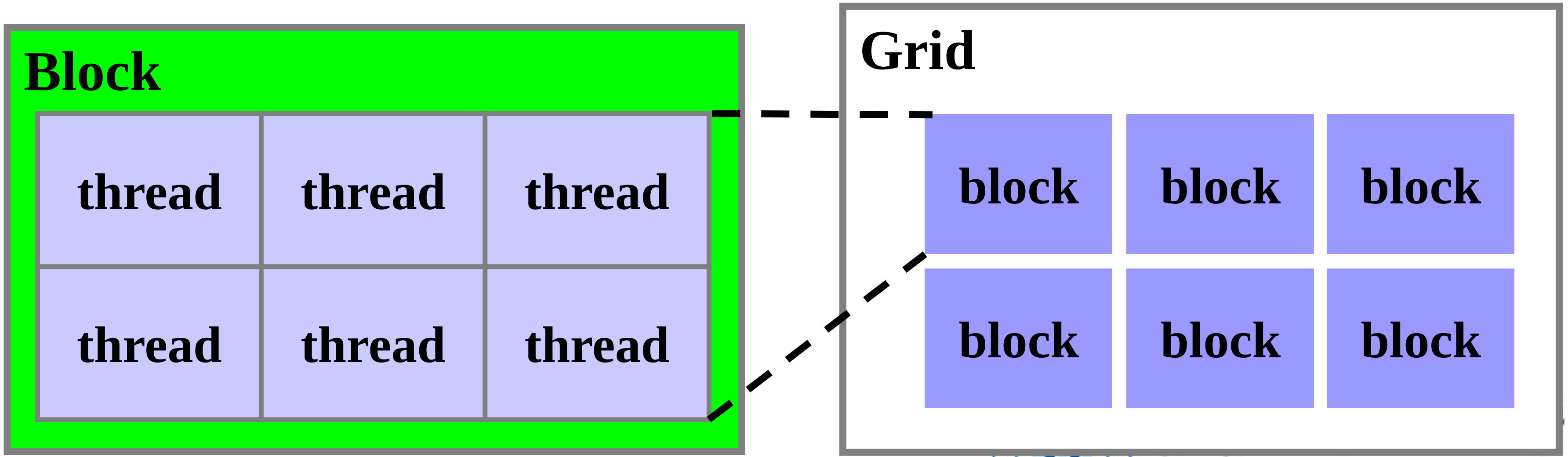


Memory Hierarchy



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



Contents

1. Programming Model
2. CUDA Language
3. Example Code Study
4. CPU & GPU Synchronization
5. Multi-GPU
6. Dynamic Parallelism

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

- Build-in device variables

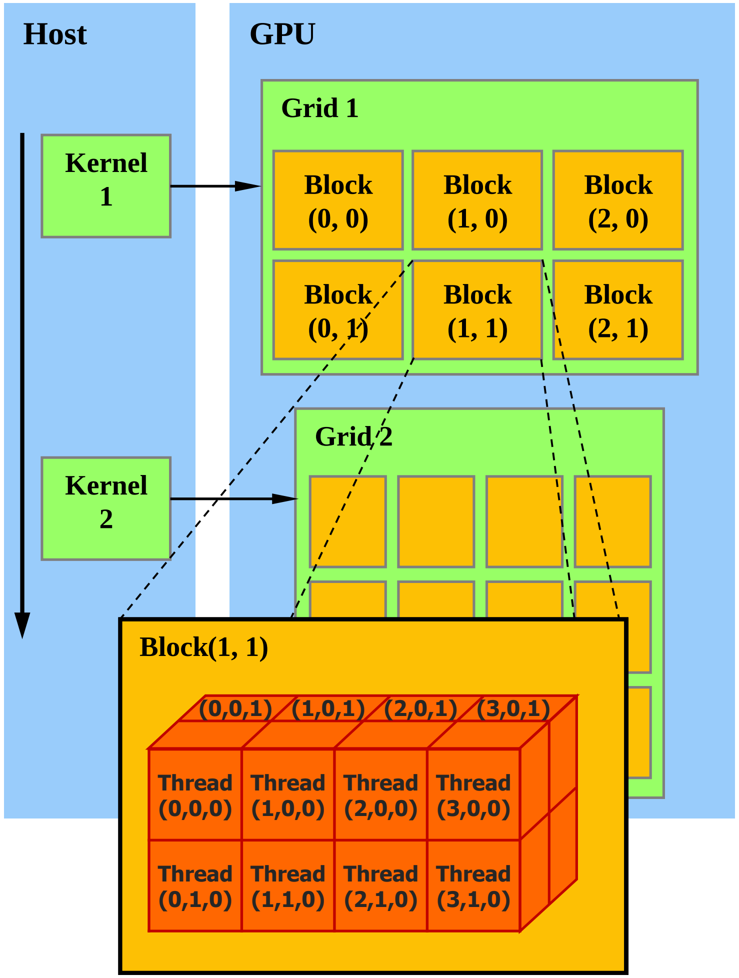
  threadIdx; blockIdx; blockDim; gridDim

- The index of threads and blocks can be denoted by a 3 dimensional struct
  - dim3 defined in vector_types.h

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

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

- Each thread can be uniquely identified by a tuple of index (x, y) or (x, y, z)



北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Function Qualifiers

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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Variable Type Qualifiers

Variable qualifiers	limitations
`__device__ var`	• Resides in device's global memory space
`__constant__ var`	• 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
`__shared__ var`	• 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

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

cudaMemcpyKind

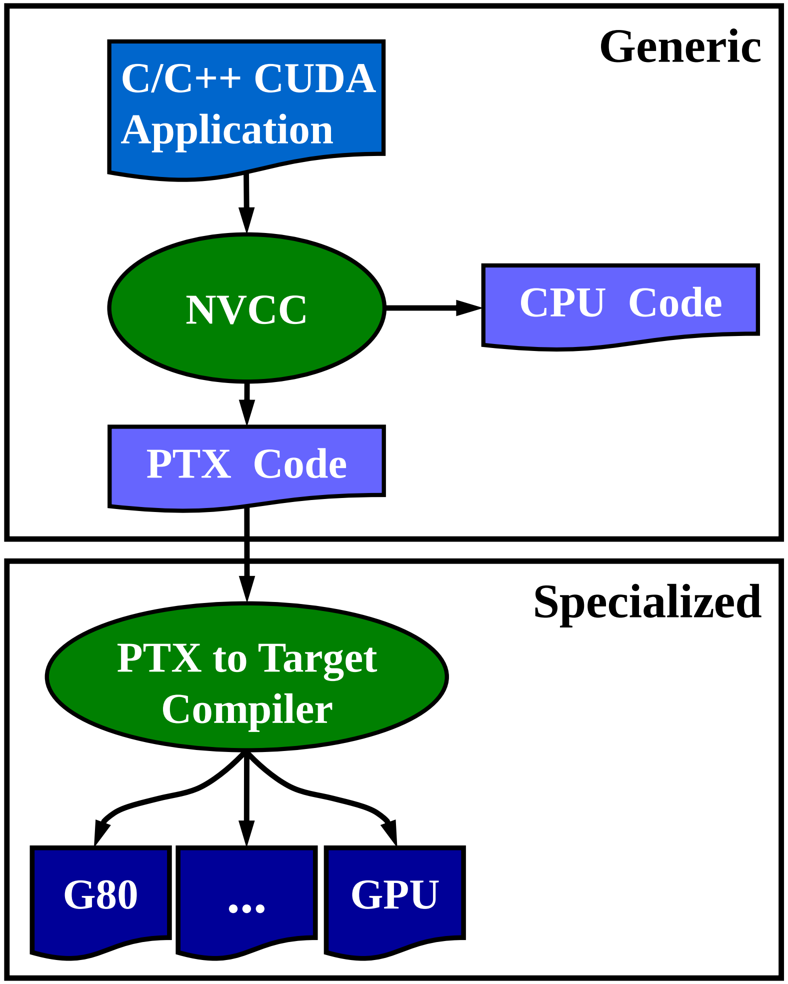
- **one of the following four value1s**

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 complication:
  - ➢ Virtual ISA (Instruction Set Architecture)
    - PTX: Parallel Threads eXecutions
  - ➢ Device-specific binary object



Contents

1. **Programming Model**
2. **CUDA Language**
3. **Example Code Study**
4. **CPU & GPU Synchronization**
5. **Multi-GPU**
6. **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;          // a, b, c的主机端备份
   int *d_a, *d_b, *d_c;     // a, b, c的设备端备份
   // 为变量a, b, c分配设备端内存
   cudaMalloc((void **)&d_a, sizeof(int));
   cudaMalloc((void **)&d_b, sizeof(int));
   cudaMalloc((void **)&d_c, sizeof(int));
   // 将输入拷贝到设备端
   cudaMemcpy(d_a,&a,sizeof(int),cudaMemcpyHostToDevice);
   cudaMemcpy(d_b,&b,sizeof(int),cudaMemcpyHostToDevice);
   // 在GPU上启动核函数add()
   add<<<1,1>>>(d_a, d_b, d_c);
   // 将结果拷贝回主机端
   cudaMemcpy(&c,d_c,sizeof(int),cudaMemcpyDeviceToHost);
   // 释放内存
   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>>>(Ad, Bd, Cd);
    ...
}
```

Alternative Implementation

- Using parallel thread instead

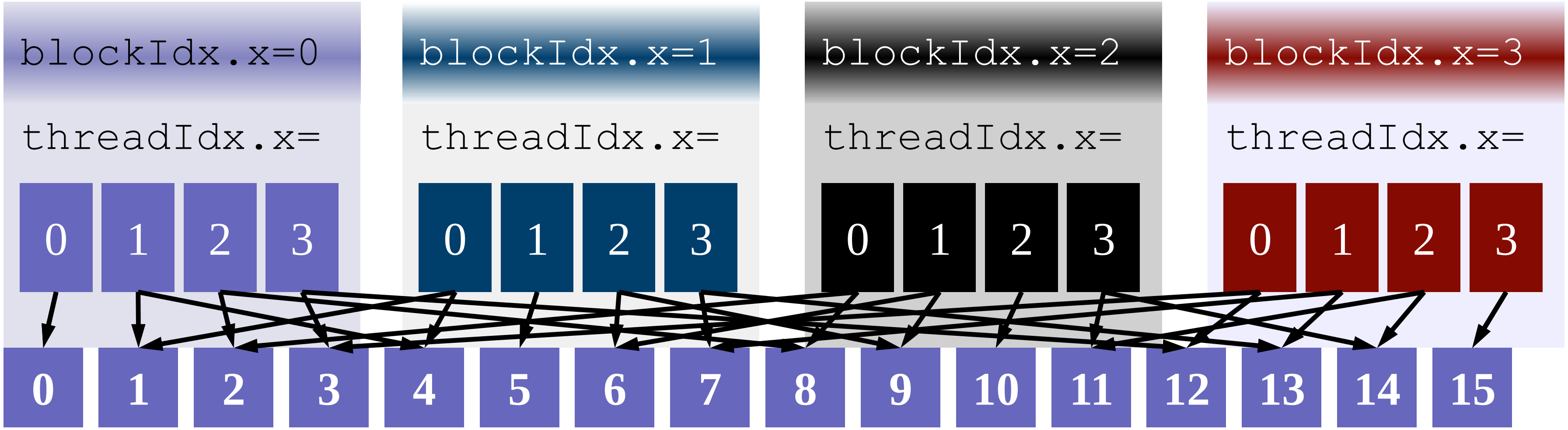
```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}

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.

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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: index = blockIdx.x * 4 + threadIdx.x
  - Method 2: index = threadIdx.x * 4 + blockIdx.x

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)

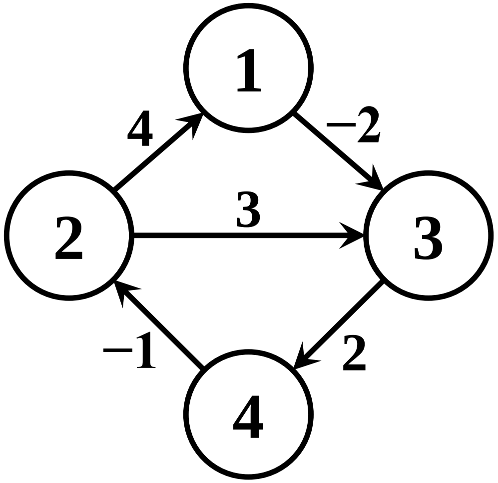
An Even More General Case

```
__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…

Example4: APSP (All Pair Shortest Paths)

- 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

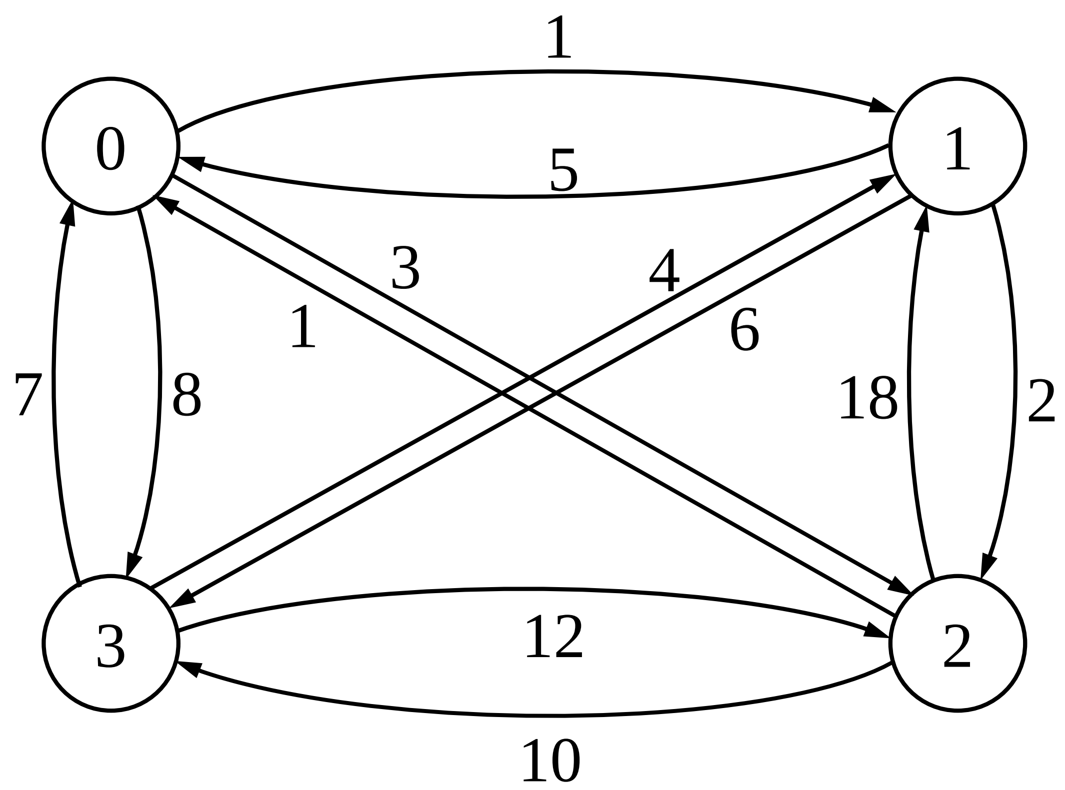
Initial weight

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

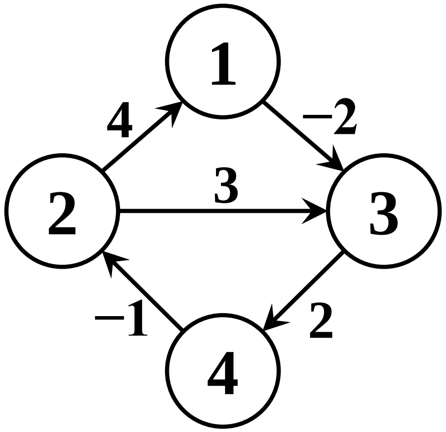
Final result

A Four-City TSP

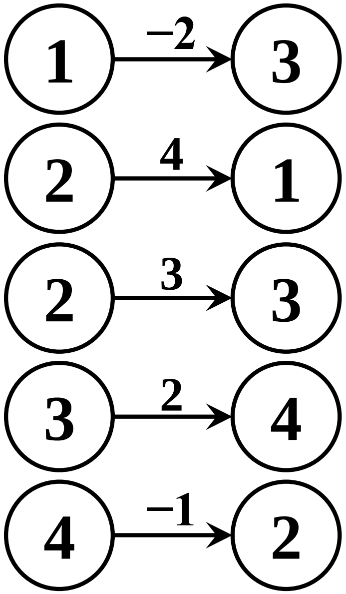
$$\begin{bmatrix} 0 & 1 & 3 & 8 \\ 5 & 0 & 2 & 6 \\ 1 & 18 & 0 & 10 \\ 7 & 4 & 12 & 0 \end{bmatrix}$$



有向非完全图——求最短路径APSP (All Pair Shortest Paths)



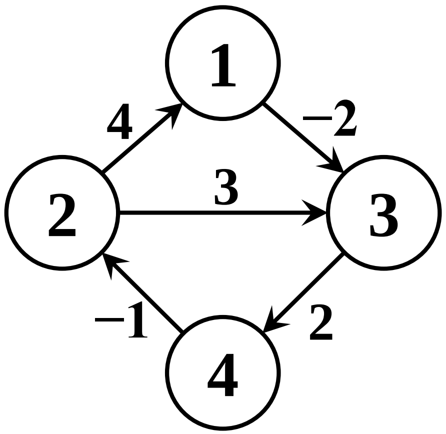
	1	2	3	4
1	0	∞	-2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	-1	∞	0



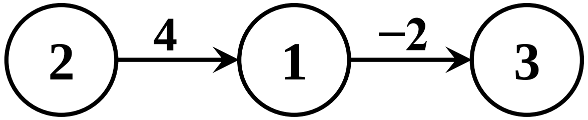
北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

假设所有路径都必须经过第1点时 (k = 1)

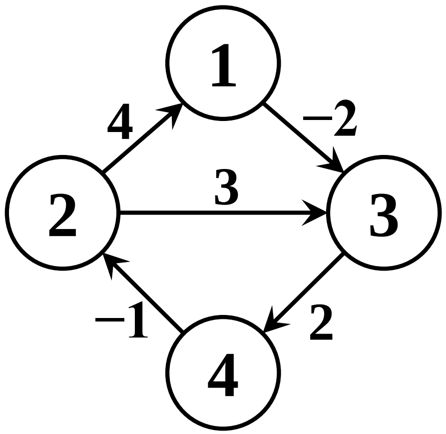
```
//经过1号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(e[i][j]>e[i][1])+e[1][j]
            e[i][j]=e[i][1]+e[1][j];
```



	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	-1	∞	0

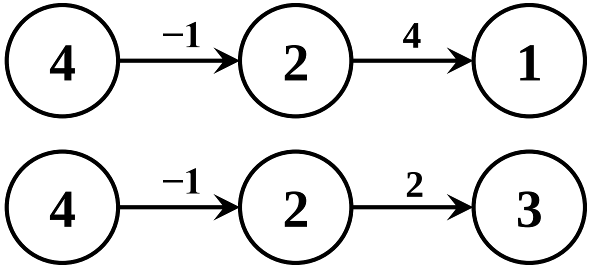


假设所有路径都必须经过第2点时 (k = 2)

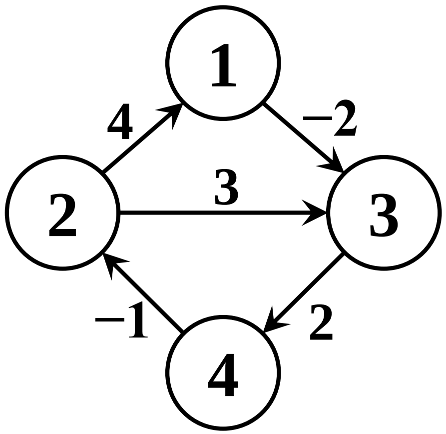



```
//经过1号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(e[i][j]>e[i][1])+e[1][j]
            e[i][j]=e[i][1]+e[1][j];

//经过2号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(e[i][j]>e[i][2])+e[2][j]
            e[i][j]=e[i][2]+e[2][j];
```

	1	2	3	4
1	0	∞	-2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	-1	1	0

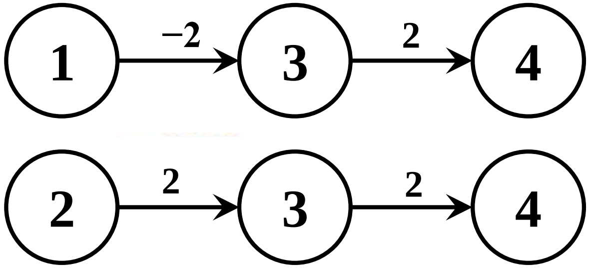


假设所有路径都必须经过第3点时 (k = 3)



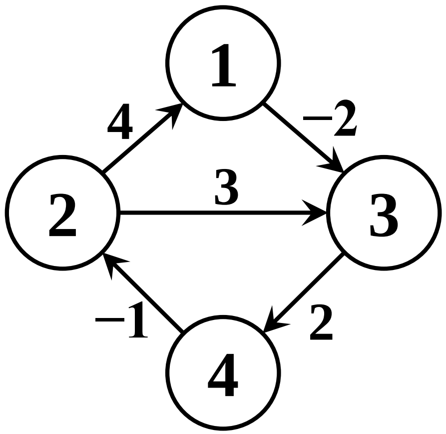
	1	2	3	4
1	0	∞	-2	0
2	4	0	2	4
3	∞	∞	0	2
4	3	-1	1	0

```
//经过1号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(e[i][j]>e[i][1])+e[1][j]
            e[i][j]=e[i][1]+e[1][j];

//经过2号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(e[i][j]>e[i][2])+e[2][j]
            e[i][j]=e[i][2]+e[2][j];

//经过3号顶点
for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
        if(e[i][j]>e[i][3])+e[3][j]
            e[i][j]=e[i][3]+e[3][j];
```



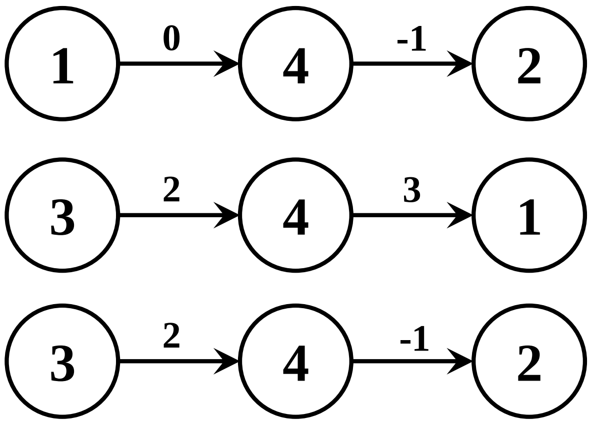
假设所有路径都必须经过第4点时 (k = 4)

How to parallelize it?




```
//弗洛伊德算法串行代码
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(e[i][j]>e[i][k])+e[k][j]
                e[i][j]=e[i][k]+e[k][j];
```

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



Implementation 1

- 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?

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

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 is 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);
}
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Contents

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Asynchronous Functions

- To facilitate concurrent execution between host and device, most CUDA function calls are asynchronous:
  - Control is returned to the host thread before the device has completed the requested task.
  - **But function calls from a kernel are serialized on GPU**
- Asynchronous functions:
  - Kernel launches
  - Asynchronous memory copy and set options: **cudaMemcpyAsync, cudaMemsetAsync**
  - cudaMemcpy within the same device
  - H2D cudaMemcpy of 64kB or less
  - cudaEvent functions

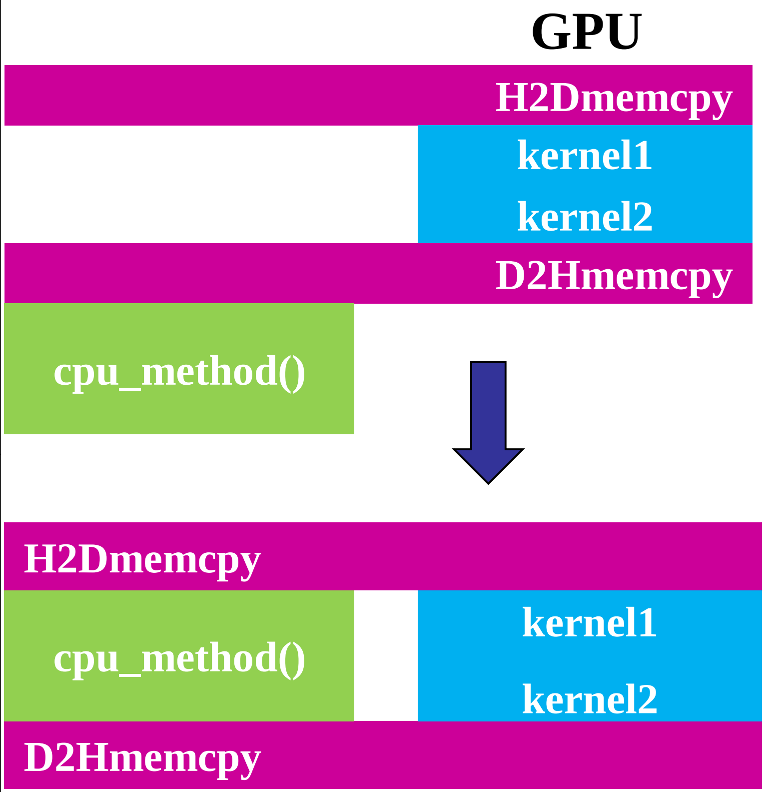
北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Why Use Asynchronous Functions?

- Overlap CPU computation with the GPU computation or data transfer

```
void main() {
   cudaMemcpy(/**/, H2D);
   kernel1<<<grid, block>>>();
   kernel2<<<grid, block>>>();
   cudaMemcpy(/**/, D2H);
   cpu_method();
}
```

```
void main() {
   cudaMemcpy(/**/, H2D);
   kernel1<<<grid, block>>>();
   kernel2<<<grid, block>>>();
   cudaMemcpyAsync(/**/, D2H);
   cpu_method();
}
```



Risk of Using Asynchronous Functions

- Programmer must enforce synchronization between GPU and CPU when there is **data dependency**

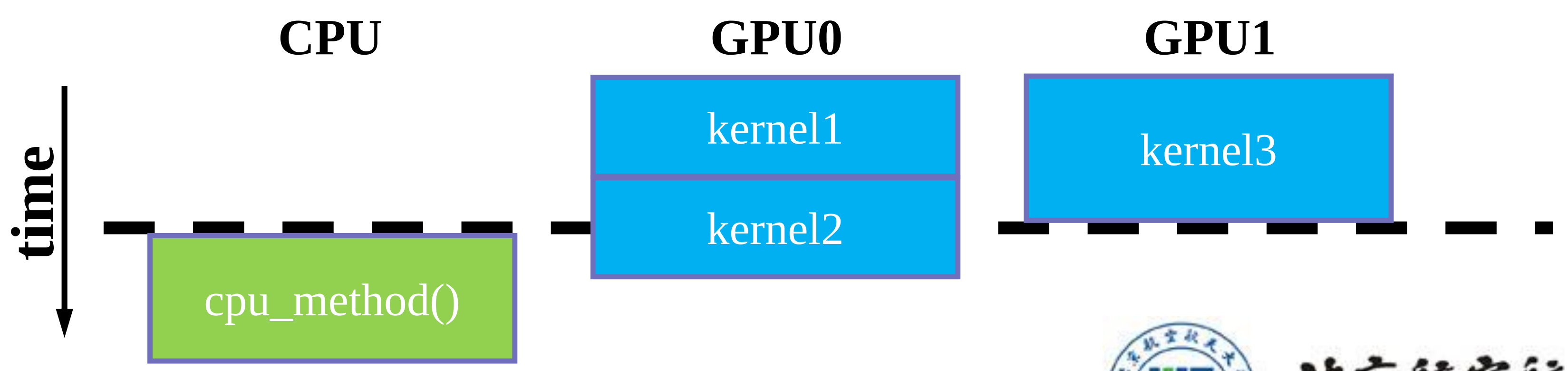
```
void main() {
    cudaMemcpyAsync(d_a, h_a, count, H2D);
    kernel<<<grid, block>>>(d_a);
    cudaMemcpyAsync(h_a, d_a, count, D2H);
    cpu_method(h_a);
}
```

Synchronization between CPU & GPU

- Device based: `cudaDeviceSynchronize()`
  - Block a CPU thread until all issued CUDA calls to a device complete
- Context based: `cudaThreadSynchronize()`
  - Block a CPU thread until all issued CUDA calls from the thread complete
- Stream based: `cudaStreamSynchronize(stream-id)`
  - Block a CPU thread until all CUDA calls in stream stream-id complete
- Event based:
  - `cudaEventSynchronize(event)`
    - Block a **CPU thread** until event is recorded
  - `cudaStreamWaitEvent(steam-id, event)`
    - Block a **GPU stream** until event reports completion

Device Synchronization Example

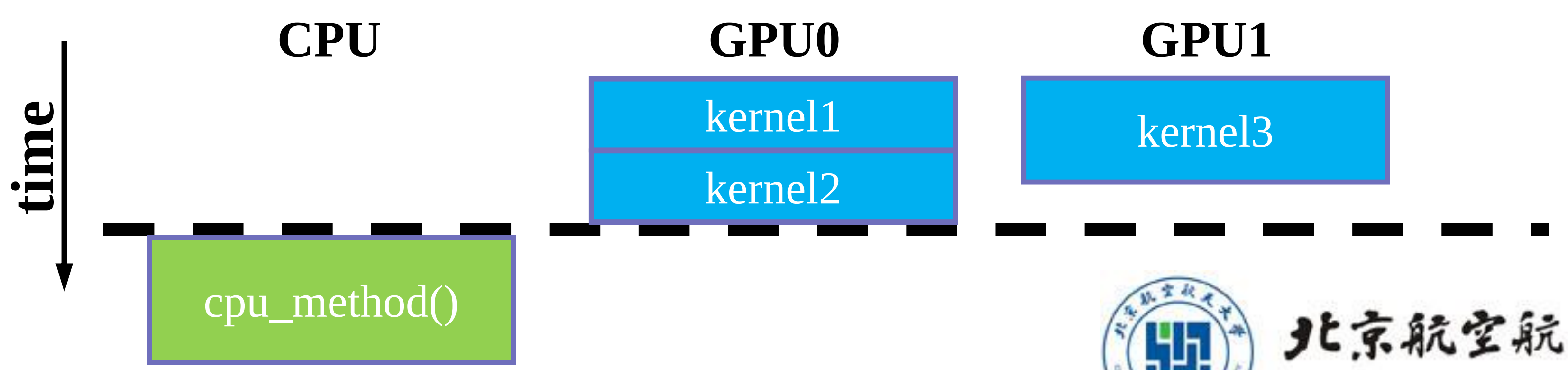
```
void main() {
   cudaSetDevice(0);
   kernel1<<<grid, block>>>();
   kernel2<<<grid, block>>>();
   cudaSetDevice(1);
   kernel3<<<grid, block>>>();
   cudaDeviceSynchronize();
   cpu_method();
}
```



Thread Synchronization Example

```
void main() {
   cudaSetDevice(0);
   kernel1<<<grid, block>>>();
   kernel2<<<grid, block>>>();
   cudaSetDevice(1);
   kernel3<<<grid, block>>>();
   cudaThreadSynchronize();
   cpu_method();
}
```

deprecated



CUDA Event

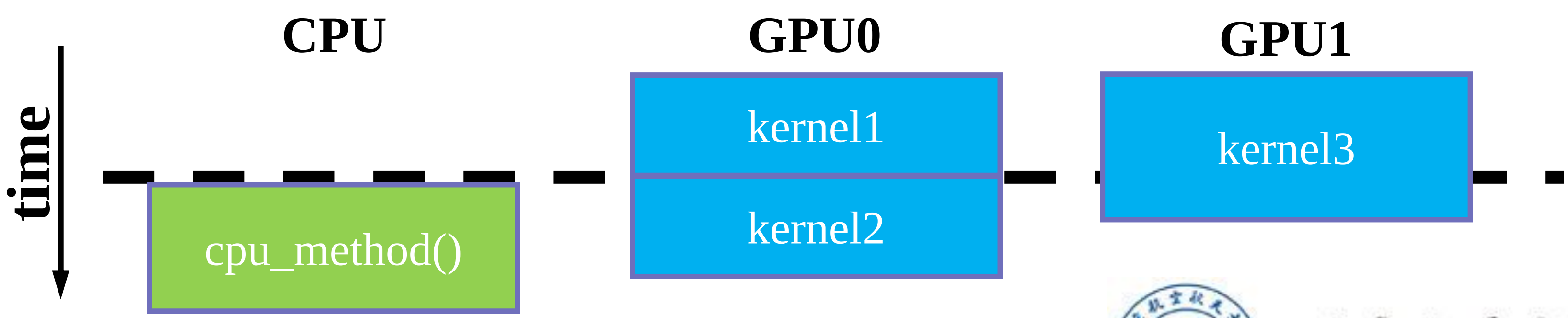
CUDA中Event主要用于在流的执行中添加标记点，用于检查正在执行的流是否到达给定点。

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()

Event Synchronization Example

```
void main() {
    cudaSetDevice(0);
    kernel1<<<grid, block>>>();
    cudaEventRecord(event)
    kernel2<<<grid, block>>>();
    cudaSetDevice(1);
    kernel3<<<grid, block>>>();
    cudaEventSynchronize(event)
    cpu_method();
}
```



Kernel Time Measurement Example

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);
kernel<<<block, thread>>>();
cudaEventRecord(stop);
cudaEventSynchronize(stop);
float time;
cudaEventElapsedTime(&time, start, stop);
```

Create event

Record event

Record event and synchronize

Compute the event duration

Contents

1. Programming Model
2. CUDA Language
3. Example Code Study
4. CPU & GPU Synchronization
5. Multi-GPU
6. Dynamic Parallelism

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Multi-GPUs

- Multi-GPUs within a node
  - A single CPU thread, multiple GPU
  - Multiple CPU threads belonging to the same process, such as pthread or openMP

- Multiple GPUs on multiple nodes
  - Need to go through network API, such as MPI

Single Thread Multi-GPUs

- All CUDA calls are issued to the current GPU
  - **cudaSetDevice()** sets the current GPU

```
// Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetNumDevices(&numDevs);
for (int d = 0; d < numDevs; d++) {
    cudaSetDevice(d);
    kernel<<<blocks, threads>>>(args);
}
```

- Asynchronous calls (kernels, memcopies) don’t block switching the GPU

```
cudaSetDevice(0);
kernel<<<...>>>(...);
cudaSetDevice(1);
kernel<<<...>>>(...);
```

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);

// 生成和CUDA devices同等数量的CPU线程
#pragma omp parallel
{
   unsigned int cpu_thread_id = omp_get_thread_num();
   unsigned int num_cpu_threads = omp_get_num_threads();
   cudaSetDevice(cpu_thread_id);
   int gpu_id = -1;
   cudaGetDevice(&gpu_id);
   printf("CPU thread %d (of %d) uses CUDA device %d¥n",
           cpu_thread_id, num_cpu_threads, gpu_id);
  ...
}
```

北京航空航天大学 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Using CUDA with MPI

```
int main(int argc, char* argv[]){
    int my_rank, size;
    int A[32];
    int i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d¥n", my_rank, size);
    for(i = 0; i< 32; i++)
        A[i] = my_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);
 }
```

Sharing Data between GPUs

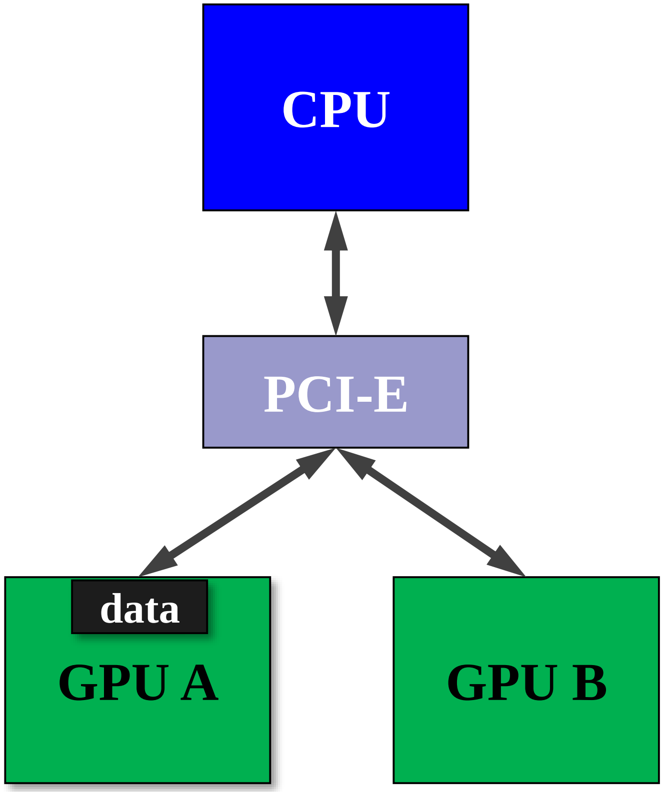
Options

1. Explicit copies via host
2. Zero-copy shared host array
3. Peer-to-peer memory copy

1. Explicit Copies via Host

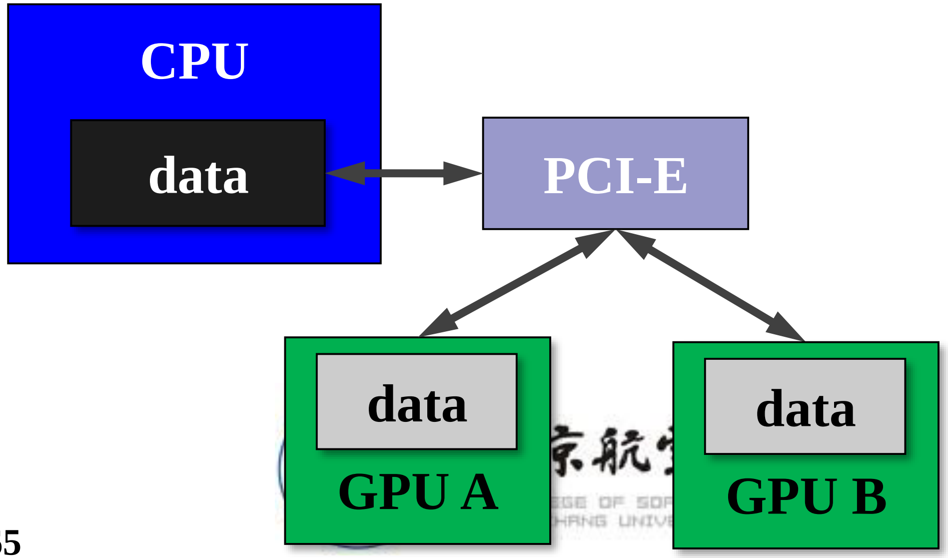
- CPU explicitly copies data from device A to device B
- Example

```
cudaSetDevice(0);
cudaMemcpy(DM1,HM,n,D2H);
cudaSetDevice(1);
cudaMemcpy(HM,DM2,n,H2D);
```



2. Using Zero-copy

- Zero-copy" refers to direct device access to host memory
- Device threads can read directly from host memory over PCI-E without using cudaMemcpy H2D or D2H
- The host memory must be pinned (page-locked)
  - Pageable memory cannot be directly accessed by the GPU because of the OS virtual memory mechanism.



Host Memory Allocation

- `malloc()`
  - Regular C library memory allocation
  - Managed by host
  - Cannot be directly accessed by GPUs
  - Must be copied to device memory via cudaMemcpy()
- `cudaMallocHost(void** hostPtr, size_t size)`
  - Allocate pinned (page-locked) host memory for higher cudaMemcpy performance
  - Used with `cudaMemAsync()` for async memory copy or CUDA stream
- `cudaHostAlloc(void** hostPtr,size_t size,`
  `unsigned int flags)`
  - Add the flag "`cudaHostAllocMapped`" to allocate pinned host memory for higher cudaMemcpy performance
  - Add the flag "`cudaHostAllocPortable`" to allocate shared host memory for "Zero copy"

→ All these functions return host memory pointer

Zero Copy via `cudaHostAlloc()`

- Allocate **host memory** using cudaHostAlloc()
  - It returns a host pointer.
  - It can enable faster host memory access.
  - Must add flag cudaHostAllocMapped for page-locked
  - Add flag cudaHostAllocPortable for sharing among all devices
- Bind host pointer to device pointer using **`cudaHostGetDevicePointer`**`(void**, void*, 0)`

```
int *hostPtr, *dev0Ptr, *dev1Ptr;
cudaHostAlloc(&hostPtr, 10, cudaHostAllocMapped|
   cudaHostAllocPortable);
cudaSetDevice(0);
cudaHostGetDevicePointer(&dev0Ptr, hostPtr, 0);
kernel<<1,10>>(dev0Ptr);
cudaSetDevice(1);
cudaHostGetDevicePointer(&dev1Ptr, hostPtr, 0);
kernel<<1,10>>(dev1Ptr);
```

Pitfalls of using Zero-copy

- PCI-E is lower in bandwidth and higher in latency than GPU global memory

    ➔ Access speed is slower than global memory

- Use zero-copy if:
    - The data is only accessed once or few times
    - Generate data on the device and copy back to host without reuse
    - Kernel(s) that access the memory are compute bound

3. Peer-to-Peer Memcpy

- Direct copy from pointer on GPU A to pointer on GPU B
- Using two functions

```
cudaError_t cudaMemcpyPeer(void *dst, int dstDevice,
const void* src, int srcDevice, size_t count)

cudaError_t cudaMemcpyPeerAsync(void *dst,int dstDevice,
const void* src, int srcDevice, size_t count,
cuda_stream_t stream=0)
```

Example: P2P memcpy

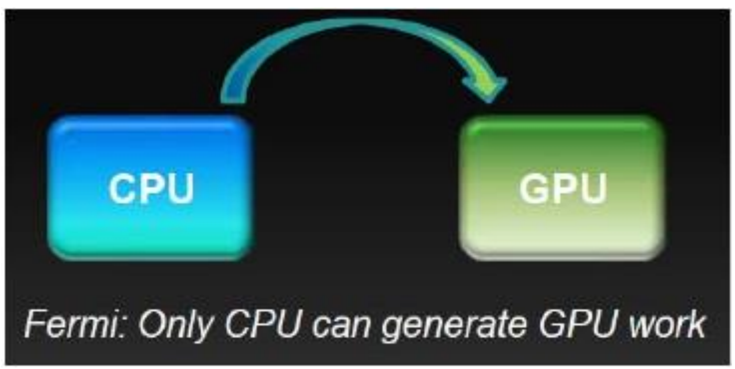
```
cudaSetDevice(0);                   //将0号设备置为当前设备
float *p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);              //在0号设备上分配内存
cudaSetDevice(1);                   //将1号设备置为当前设备
float *p1;
cudaMalloc(&p1, size);              //在1号设备上分配内存
cudaSetDevice(0);                   //将0号设备置回当前设备
Kernel1<<<1000, 128>>>(p0);         //在0号设备上启动核函数
cudaSetDevice(1);                   //将1号设备置回当前设备
cudaMemcpyPeer(p1, 1, p0, 0, size);   //将p0拷贝到p1
Kernel1<<<1000, 128>>>(p1);         //在1号设备上启动核函数
```

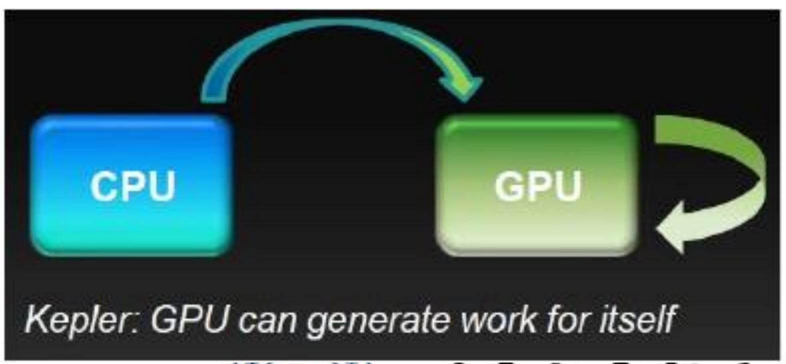
Contents

1. Programming Model
2. CUDA Language
3. Example Code Study
4. CPU & GPU Synchronization
5. Multi-GPU
6. Dynamic Parallelism

Dynamic Parallelism

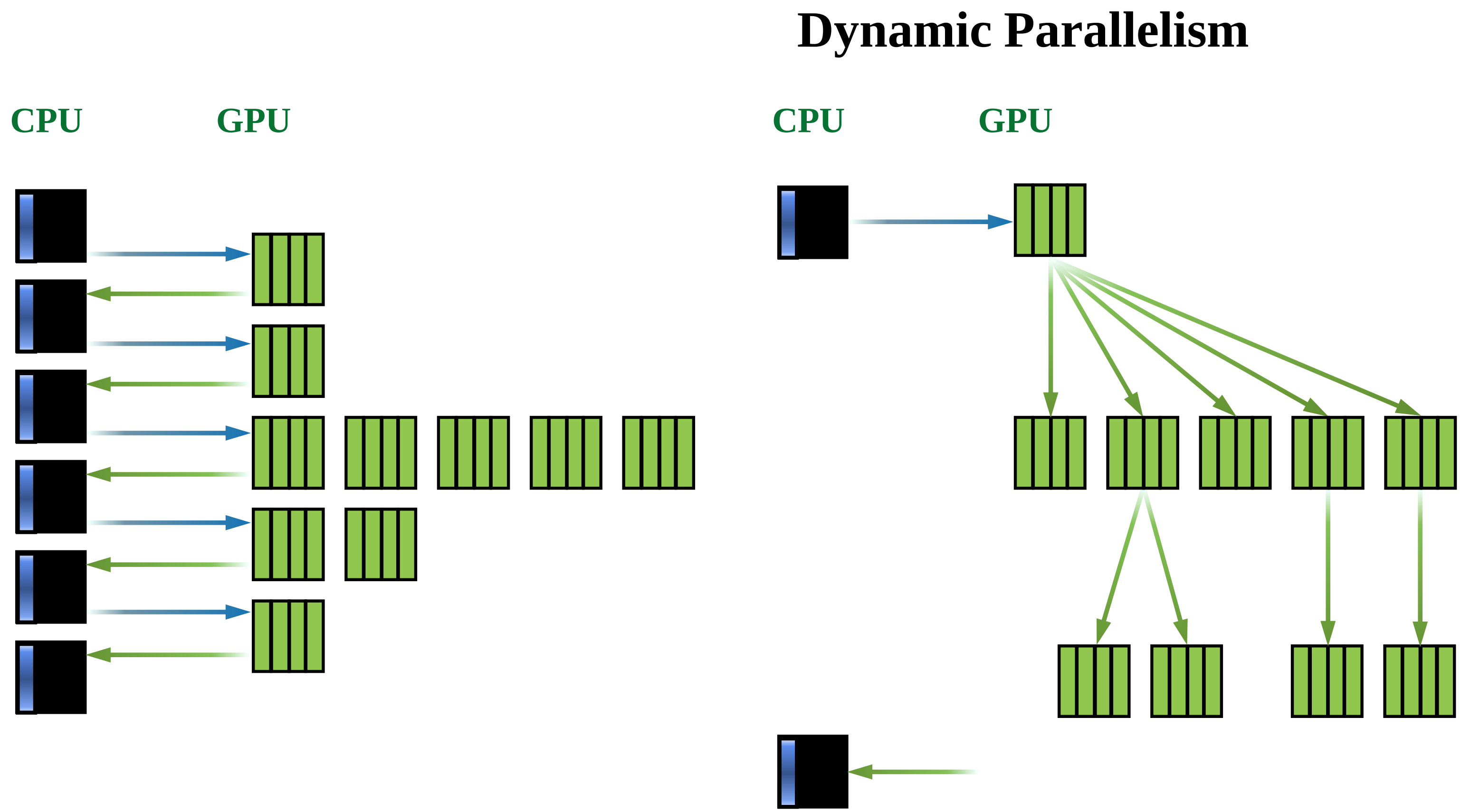
- The ability to launch new grids from the GPU
  - Dynamically
  - Simultaneously
  - Independently
- Supported from CUDA5.0 on devices of Compute Capability 3.5 or higher





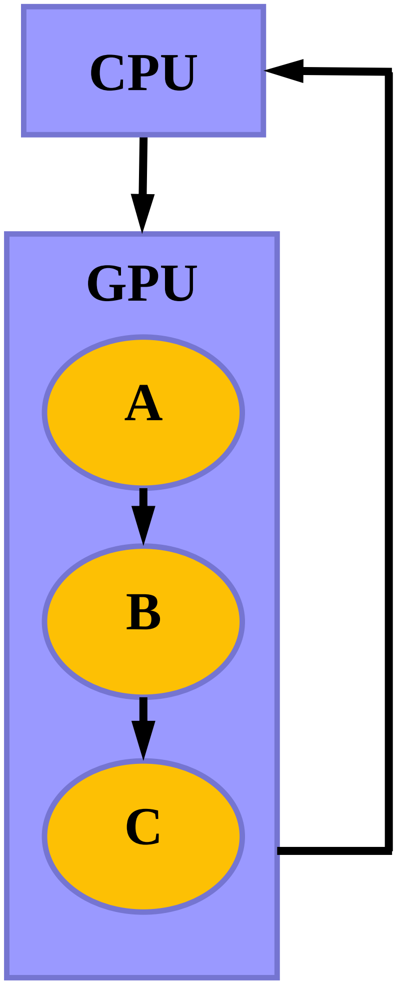
What does It Mean?

- Reduce the number of kernel launches



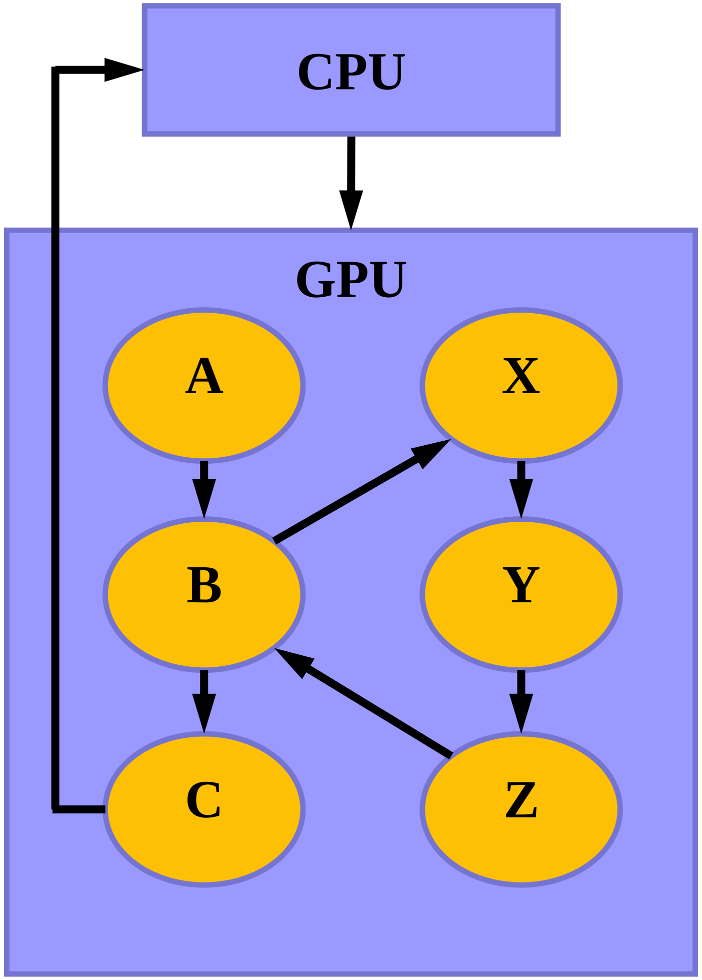
Dependency in CUDA

```
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);
}
```

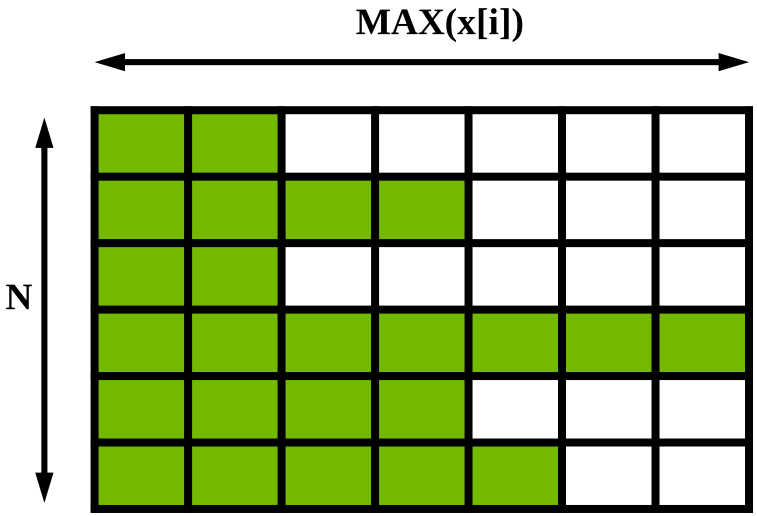


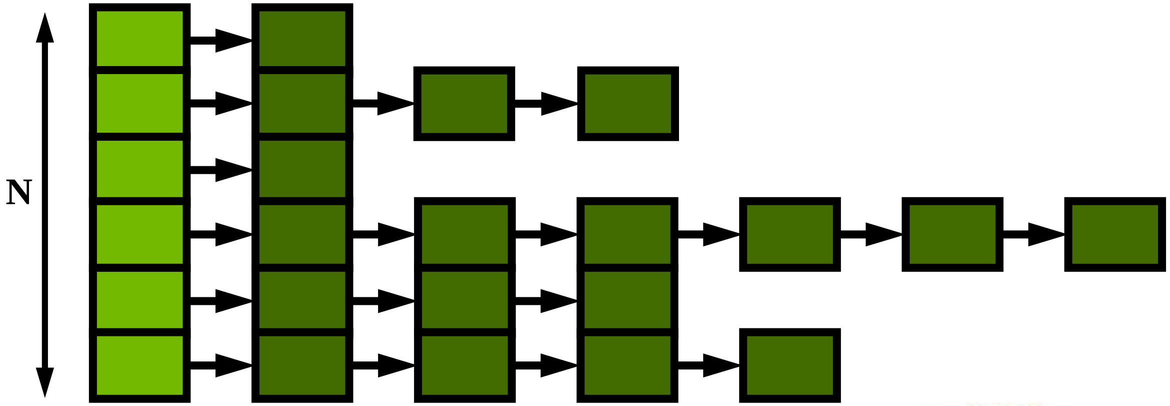
What is DP Good for?

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

1. Dynamic Block Size and Grid Size

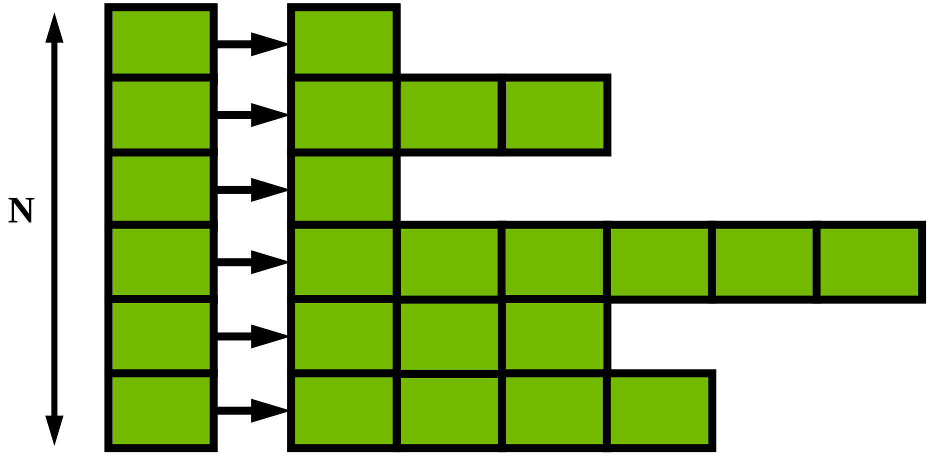
```
for i = 1 to N
   for j = 1 to x[i]
      convolution(i, j)
   next j
next i
```



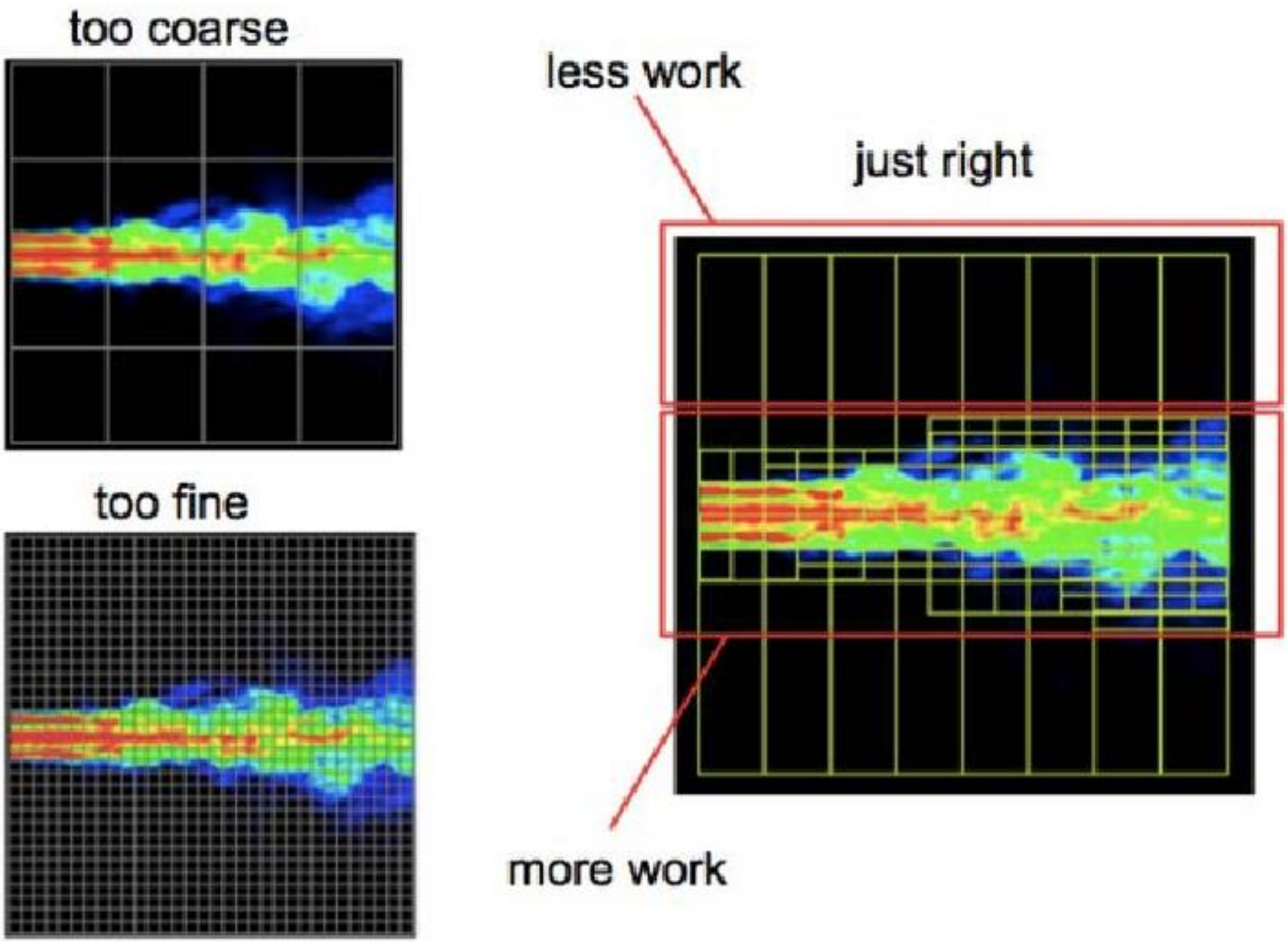


1. Dynamic Block Size with DP

```
__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)

CPU Code

```
degtrf(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
}
```

GPU Code

```
idamax();
dswap();
dscal();
dger();
dlaswap();
dtrsm();
dgemm();
```

LU decomposition (Kepler)

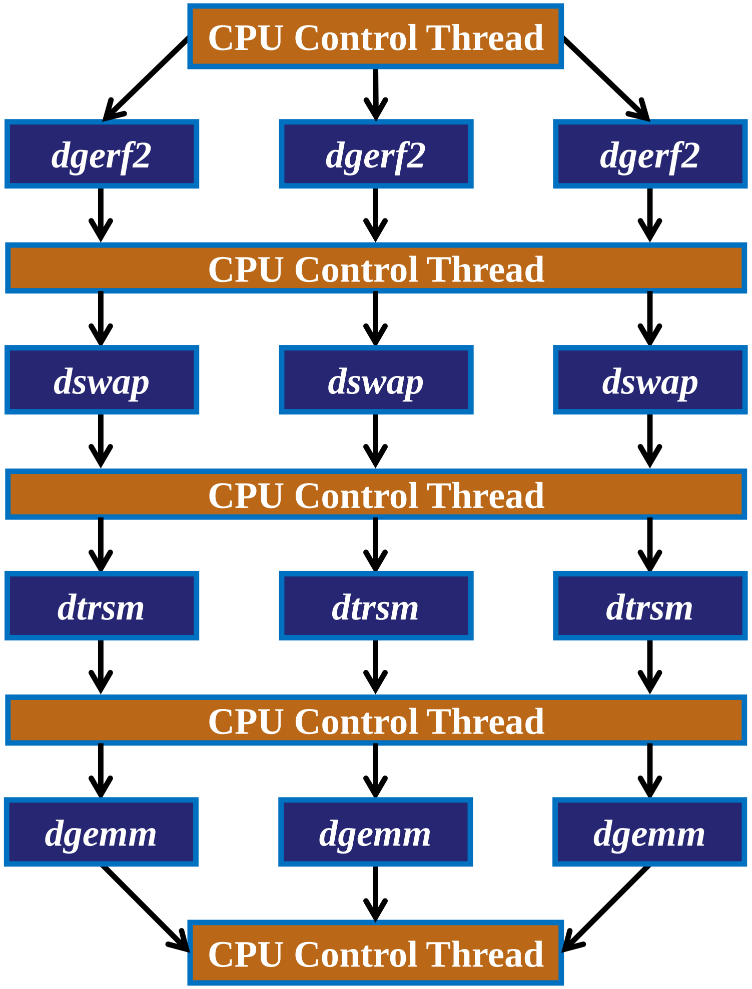
CPU Code

```
degtrf(N, N) {
  degtrf<<<>>>

  synchronize();
}
```

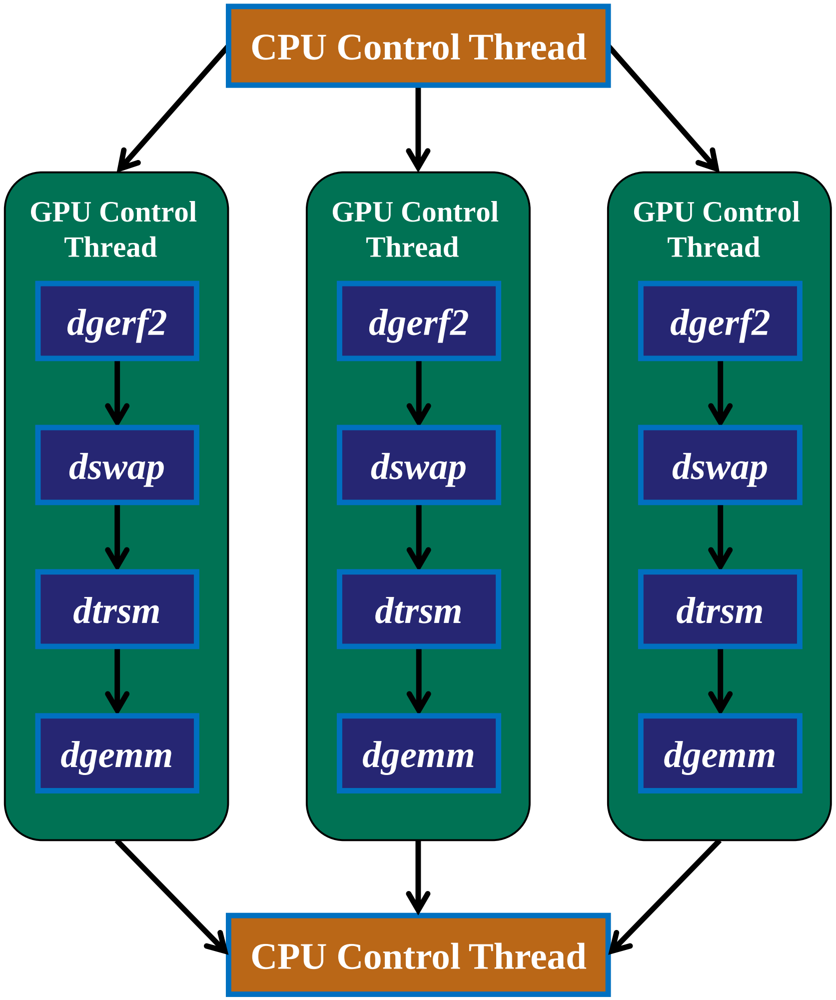
GPU Code

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

LU Decomposition



Multiple LU-Decomposition, Pre-Kepler



Batched LU-Decomposition, Pre-Kepler

4. Library Call

```
__global__ void libraryCall(float *a,float *b,float *c){
    // 所有线程都生成数据All threads generate data
    createData(a, b);
    __syncthreads();
    // Only one thread calls library
    if(threadIdx.x == 0) {
        cublasDgemm(a, b, c);
        cudaDeviceSynchronize();
    }
    // 所有线程都等待dtrsm
    __syncthreads();
    // 继续
    consumeData(c);
}
```

cuBLAS:
Cuda矩阵运算库

5. Parallel Recursion

Quick sort

```
__global__ void qsort(int *data, int left, int right)
{
    int pivot=data[0];
    int *lptr=data+left, *rptr=data+right;
    // 根据中枢值将数据一分为二
    partition(data, left, right, lptr, rptr, pivot);
    // 启动下一级递归
    if(left<(rptr-data))
        qsort<<<...>>>(data, left, rptr-data);
    if(right>(lptr-data))
        qsort<<<...>>>(data, lptr-data, right);
}
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院