An Introduction to GPU Architecture and Programming

Hao Wang
The Ohio State University

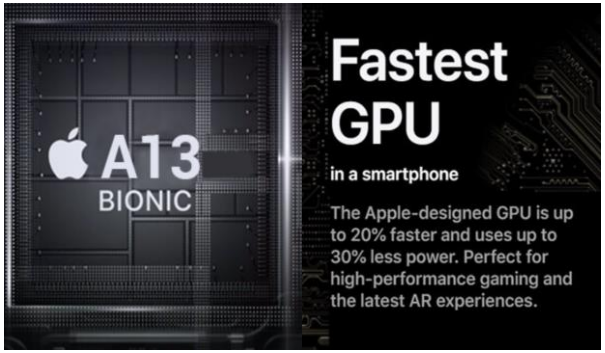
wang.2721@osu.edu
hwang121@gmail.com

Graphics Processing Units (GPUs) are Everywhere

Mobile Phones

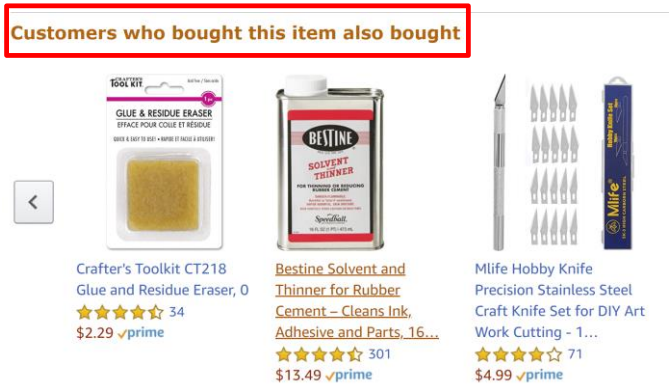
Desktops / Laptops

Data Center Severs



In September 2019, **Apple** announced their fastest GPU inside **A13 BIONIC** chip for iPhone 11, 11 pro, and 11 pro max.

In November 2019, **NVIDIA** recommended their GPUs for high detail-level (60+ FPS) game RED DEAD REDEMPTION II.



Amazon presents a list of products for **"Customers who bought this item also bought".** Algorithms of the recommendation system are running on GPUs of Amazon Cloud.

THE OHIO STATE UNIVERSITY

GPU Timeline: from GPU to GPGPU

- Graphics Processing Unit (GPU)
  - Originally designed for video decoding, gaming, visualization, etc. (1970s – 1990s)
  - Also called video cards

- **NVIDIA** GeForce 256 (in 1999)
  - "the world's first GPU"
  - Integrated transform, lighting, triangle setup/clipping, and rendering engines

Question: We already have CPU for general-purpose computing. Why do we need GPGPU?

- **NVIDIA CUDA: Compute Unified Device Architecture** (in 2007)
  - A parallel computing platform and application programming interface model
  - Programmers can code and execute algorithms on GPU

- **GPGPU**: **G**eneral-**P**urpose **GPU** (2007-)

Moore's Law and Dennard Scaling

Gordon Moore

The number of transistors in a dense integrated circuit doubles about every two years.

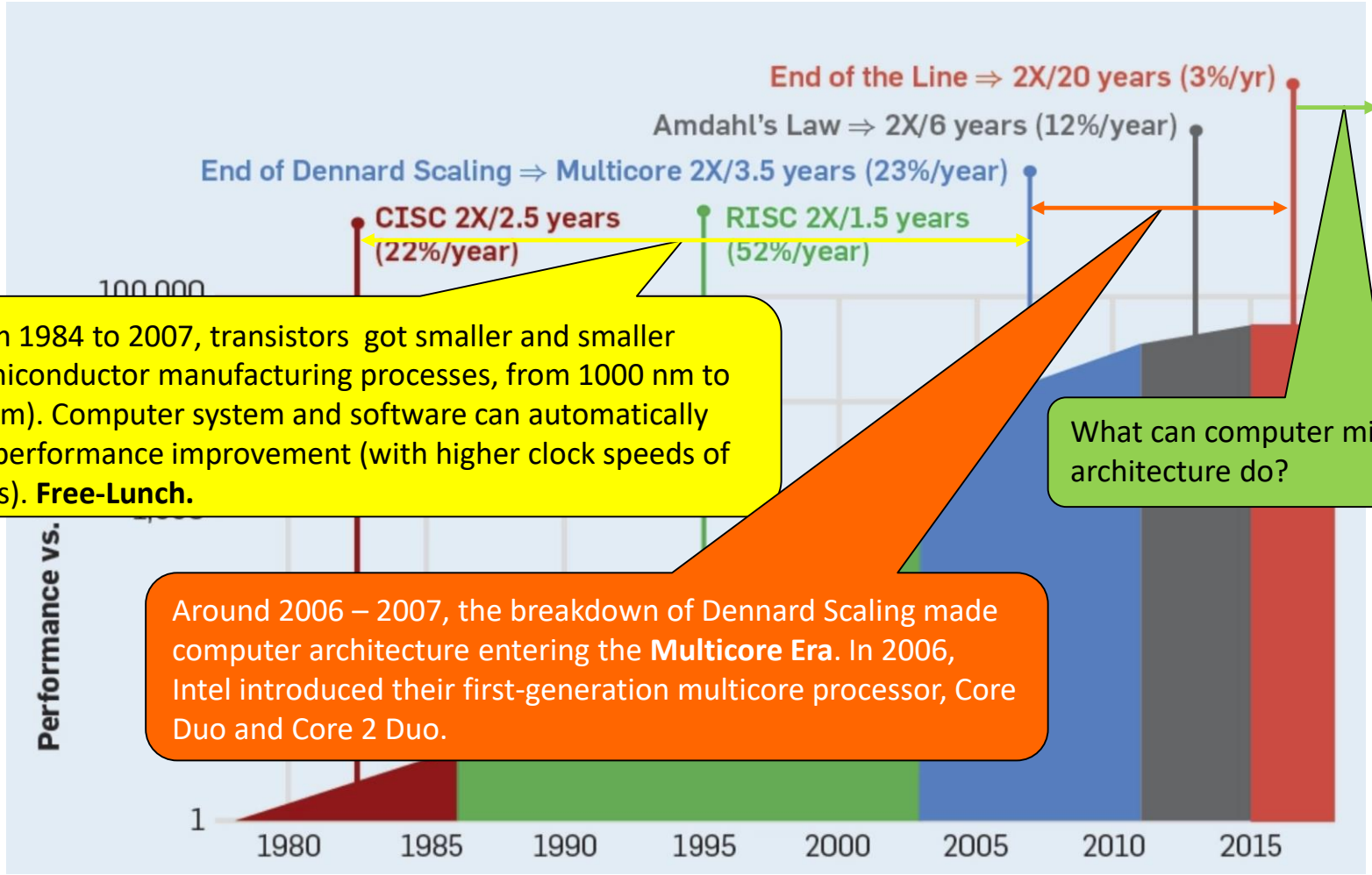
-- Gordon Moore in 1965

Robert H. Dennard

As transistors get smaller, their power density stays constant, so that the power use stays in proportion with area.

-- Robert H. Dennard in 1974

Growth of Computer Performance

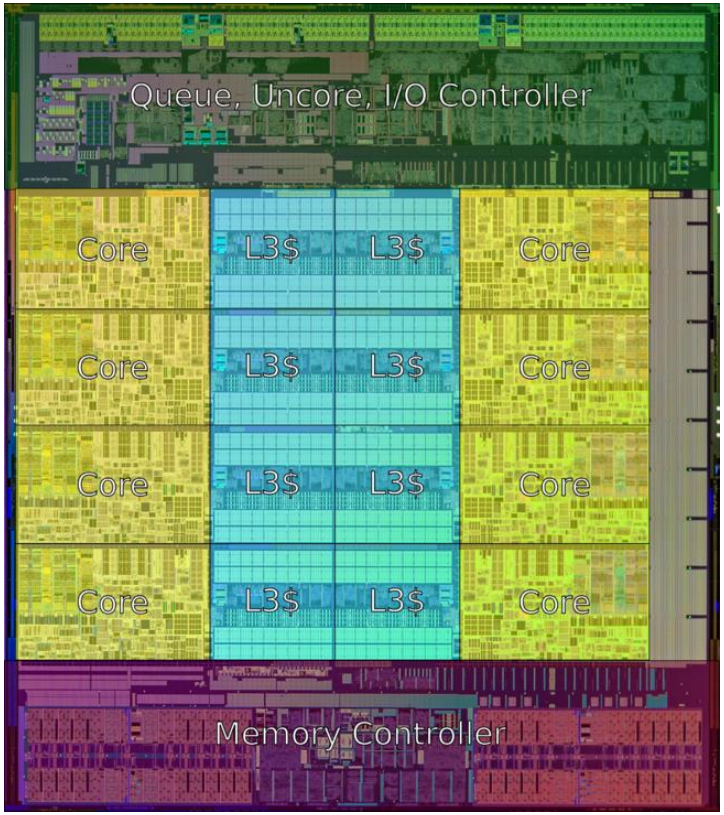


Growth of computer performance using integer programs (SPECintCPU)

John L. Hennessy, David A. Patterson, "A New Golden Age for Computer Architecture", Comm. Of the ACM, February 2019

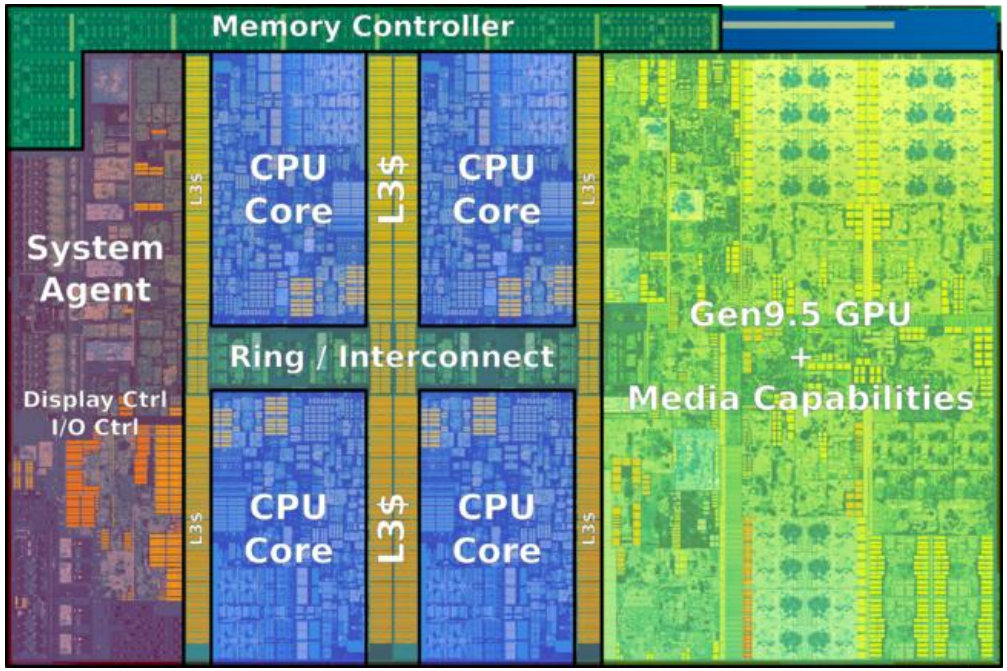
THE OHIO STATE UNIVERSITY

Intel Multicore CPUs in Past 5 Years



Intel Core i7-5960X (Haswell-E) Octa-core CPU (in 2014)

A large portion of wafer area is used for shared L3 cache, memory controller, and I/O controller.



Intel Core i7-8705G (Kaby Lake) Quad-core CPU (in 2018)

The die area is also used for the integrated graphics.

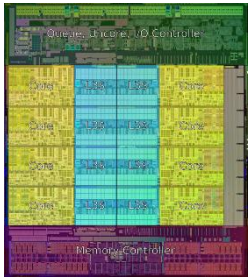
Q2: The figure in the previous slide said the growth of CPU performance is reaching an end. But, wait, how can we check CPU performance?

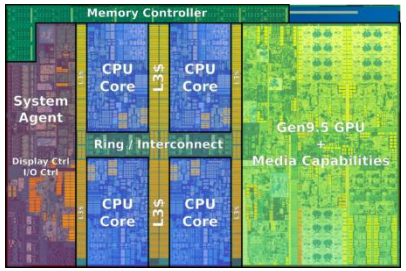
Theoretical Peak Performance

- **FLOPS** – **FL**oating-point **O**perations **P**er **S**econd
  - MFLOPS: mega FLOPS (10^6 = 1,000,000)
  - GFLOPS: giga FLOPS (10^9 = 1,000,000,000)
  - TFLOPS: tera FLOPS (10^{12} = 1,000,000,000,000)

Theoretical Peak Performance =

computing unit clock speed * vector operations per cycle * number of cores





Intel Core i7-5960X (Haswell-E) (in 2014)

- 3.0 GHz Frequency
- 16 FP64 operations per cycle (double precision)
- 8 cores

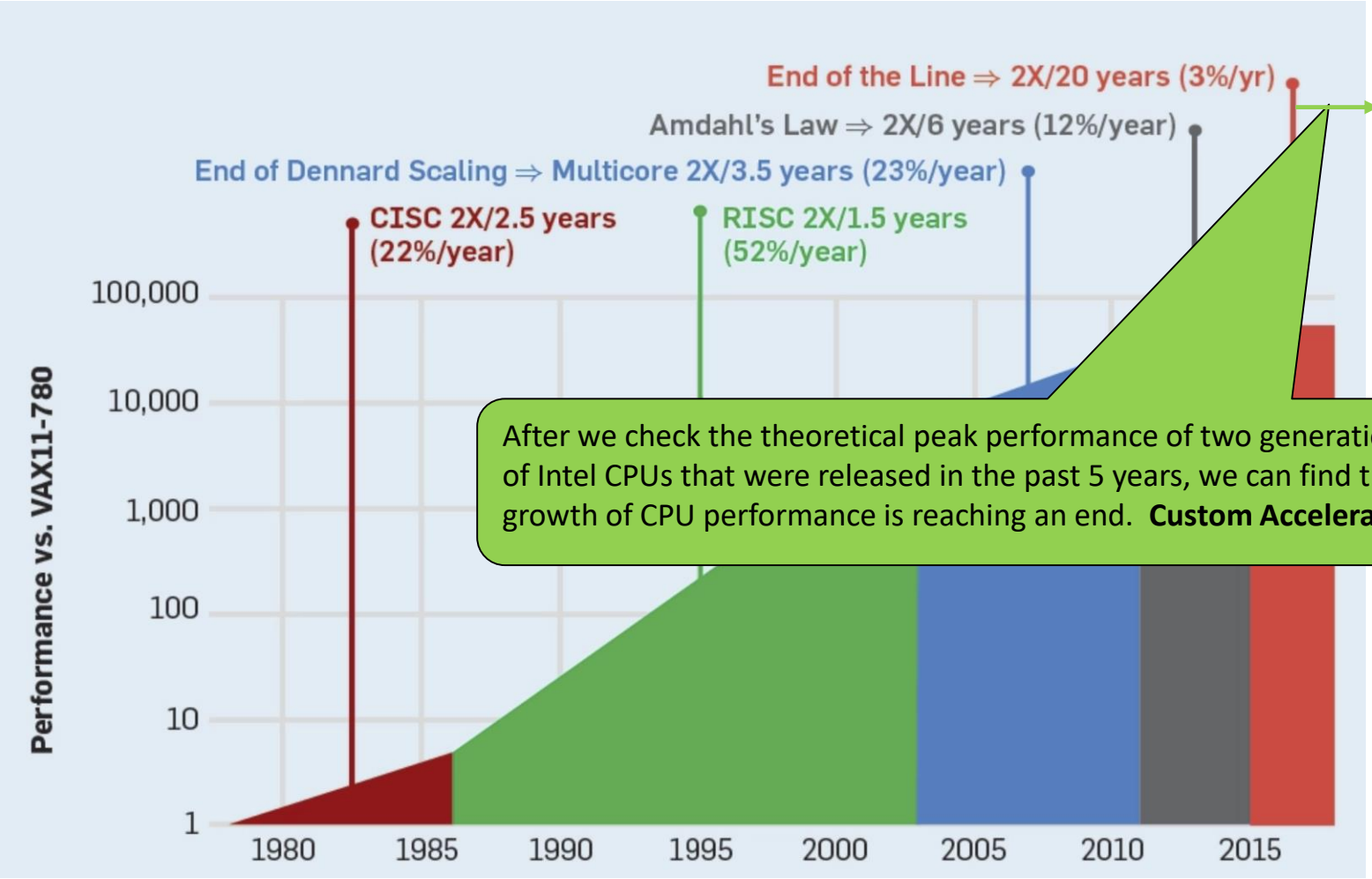
3.0 * 16 * 8 = **384 GFLOPS**

Intel Core i7-8705G (Kaby Lake) (in 2018)

- 3.1 GHz Frequency
- 16 FP64 operations per cycle (double precision)
- 4 cores

3.1 * 16 * 4 = **198.4 GFLOPS**

Growth of Computer Performance

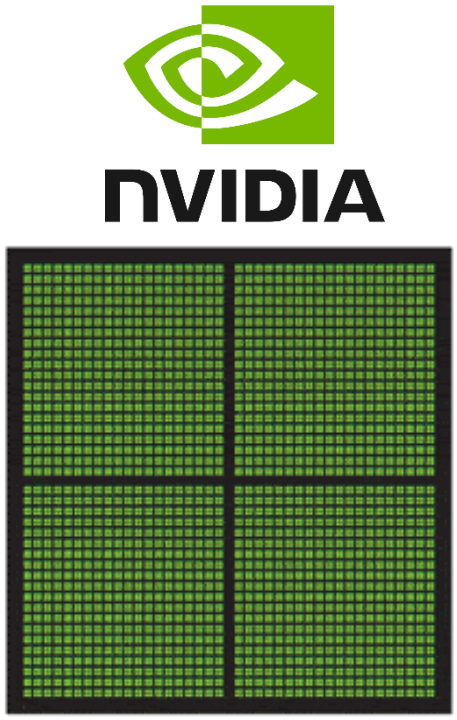


Growth of computer performance using integer programs (SPECintCPU)

John L. Hennessy, David A. Patterson, "A New Golden Age for Computer Architecture", Comm. Of the ACM, February 2019

Overview of Architectures of CPU and GPU

CPU
MULTIPLE CORES



GPU
THOUSANDS OF CORES

CPUs have **a few powerful** cores optimized for **serial and general-purpose** computing.

GPUs have **thousands of smaller and less-powerful** cores for **parallel and specific** computing.

NVIDIA GPUs

Each SM of **NVIDIA V100**:

64 INT cores

64 single-precision (FP32) cores

32 double-precision (FP64) cores

8 Tensor cores

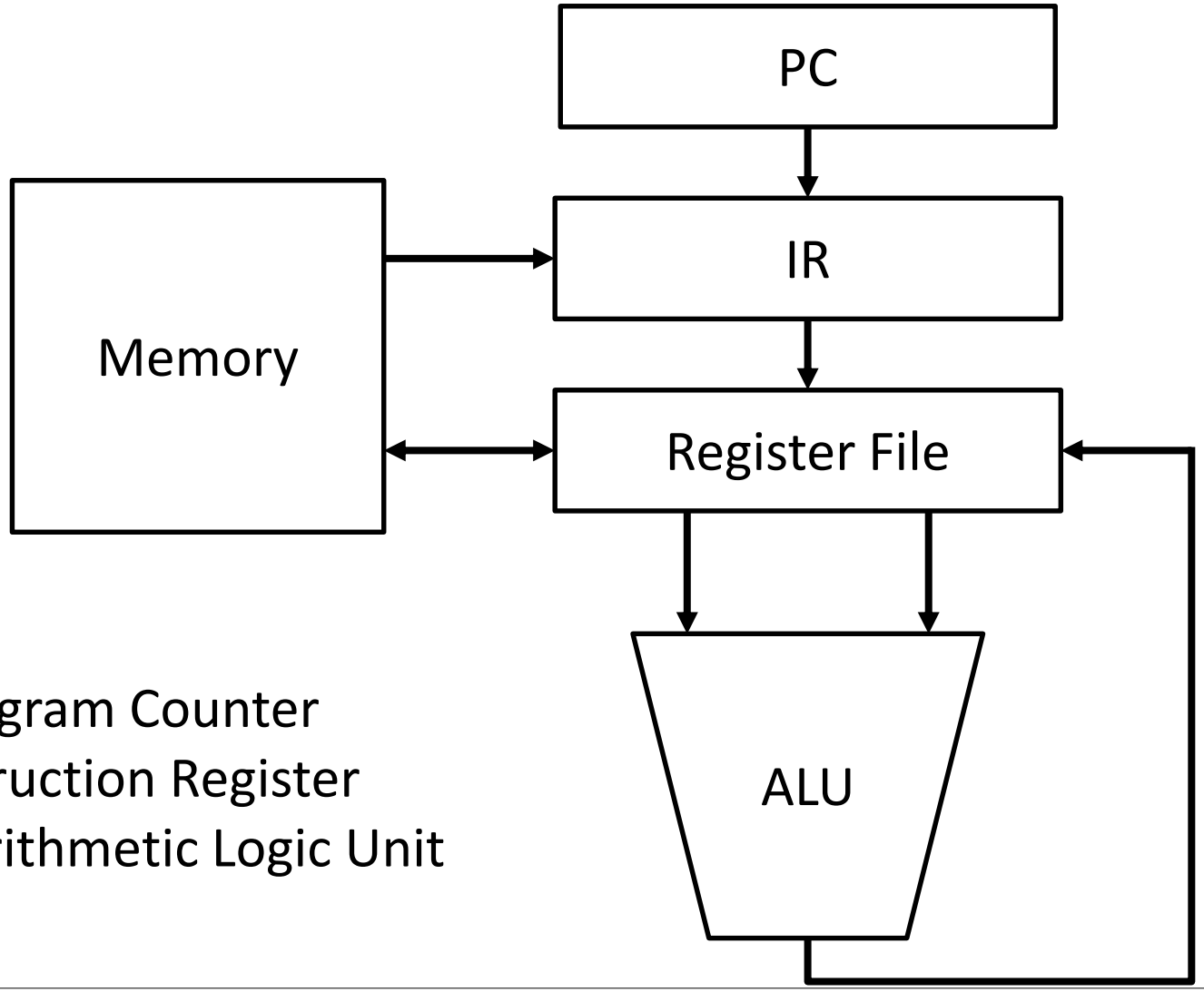
128 KB L1 data cache + shared memory

256 KB register file

- **NVIDIA Tesla V100 (Volta) GPU** (released in Dec 2017) includes
  - 80 SMs **(Streaming Multiprocessor)**
  - 5120 INT cores + 5120 FP32 cores + 2560 FP64 cores + 640 tensor cores

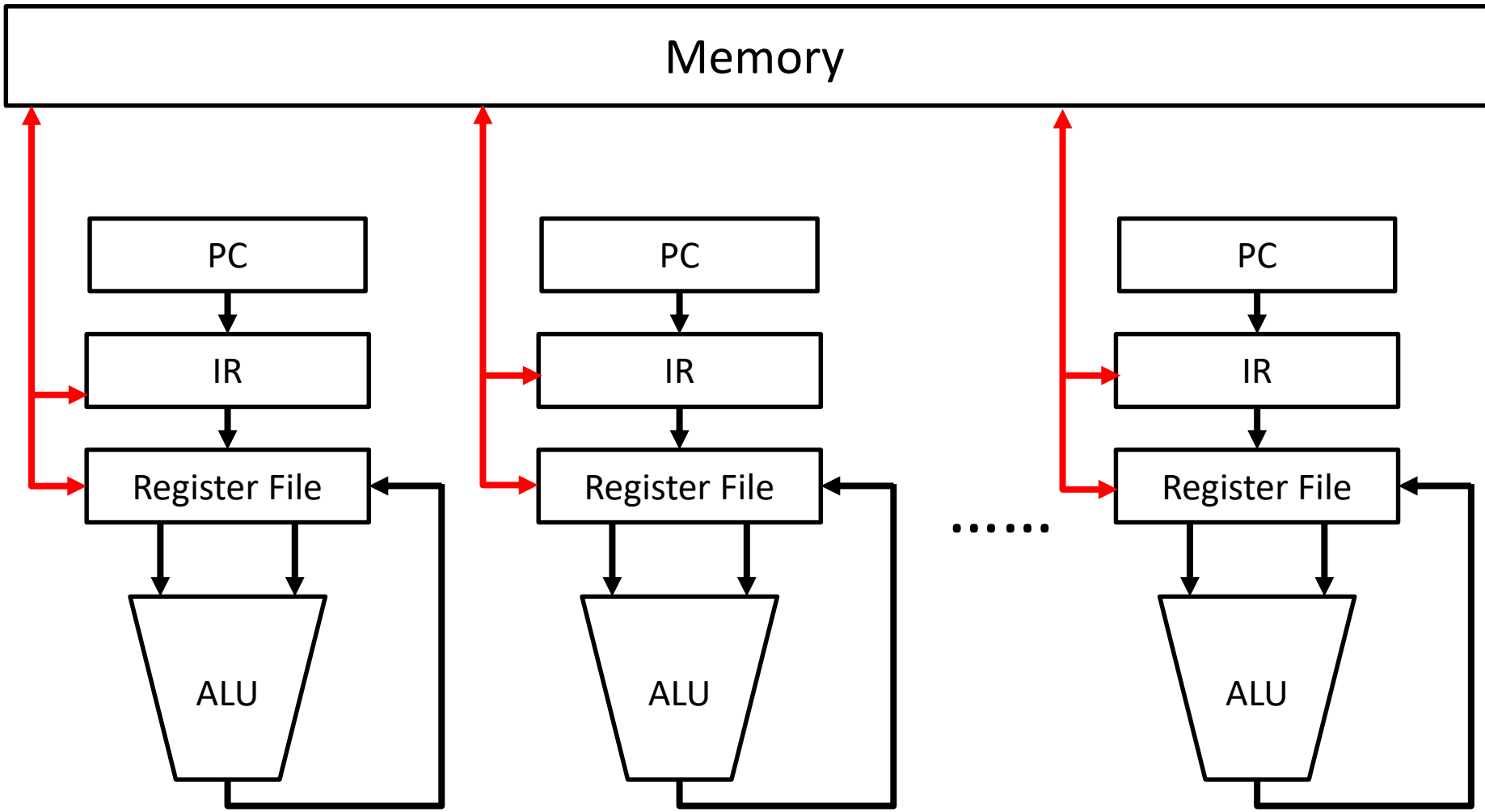
Theoretical Peak performance: 1.38 GHz * 2 * 2560 = **7.066 TFLOPS** (7.8 TFLOPS in NVIDIA Volta whitepaper is based on GPU boost clock.)

Sequential Execution Flow of CPU



PC: Program Counter
IR: Instruction Register
ALU: Arithmetic Logic Unit

Multicore CPU Architecture



- Each core has **independent** ALU and control logic units (PC, IR, etc.)
- A group of cores share memory controller and I/O controller

Manycore GPU Architecture

- Independent ALU
- **Shared control logic units** (PC, IR, Scheduler…)

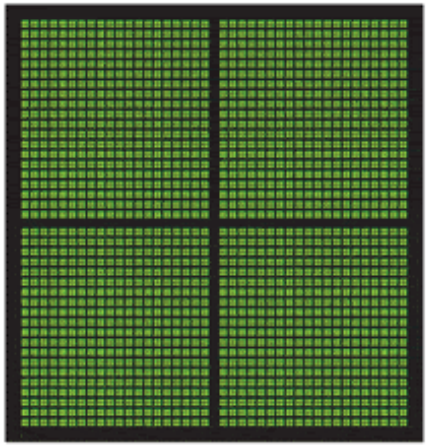
In each SM of **NVIDIA Tesla V100***, a group of cores share IR and warp scheduler.

* In earlier NVIDIA GPU architectures, e.g., Pascal and Fermi, a group of cores share a single PC; while, Volta has the per-thread performance counter architecture.

Summary – from perspective of CPU/GPU Architectures

CPU: a few powerful cores

CPU
MULTIPLE CORES



GPU
THOUSANDS OF CORES

GPU: thousands of smaller and less-powerful cores

Less cores but larger caches

- For temporal locality and spatial locality (reduce data access latency)

Sophisticated control

- Branch prediction (reduce branch latency)
- Data forwarding (reduce data access latency)

Powerful ALUs

- Reduce operation latency

More cores but smaller caches

- Boost memory throughput

Simpler control

- No branch prediction
- No data forwarding
- Shared control logic units

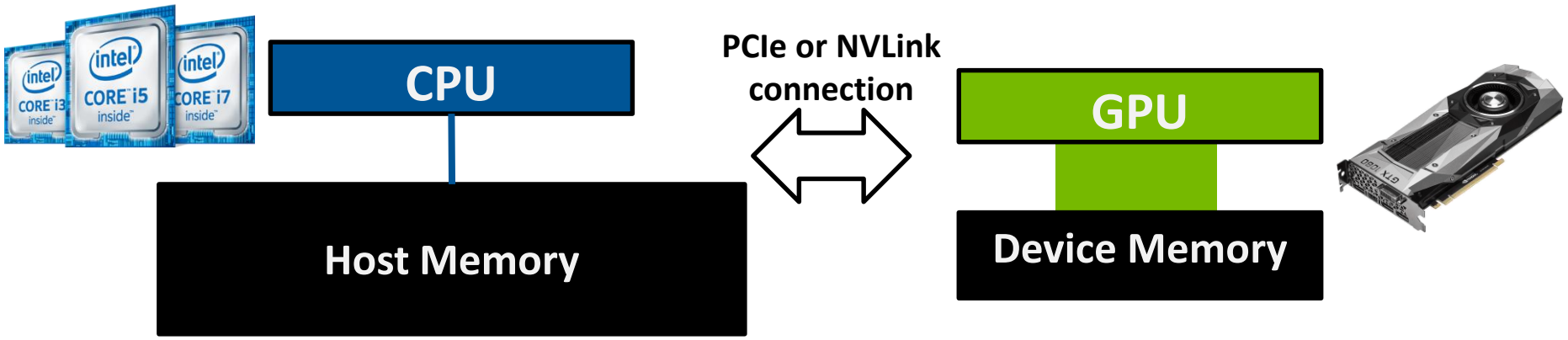
Energy-efficient ALUs

- Pipelined for high throughput

THE OHIO STATE UNIVERSITY

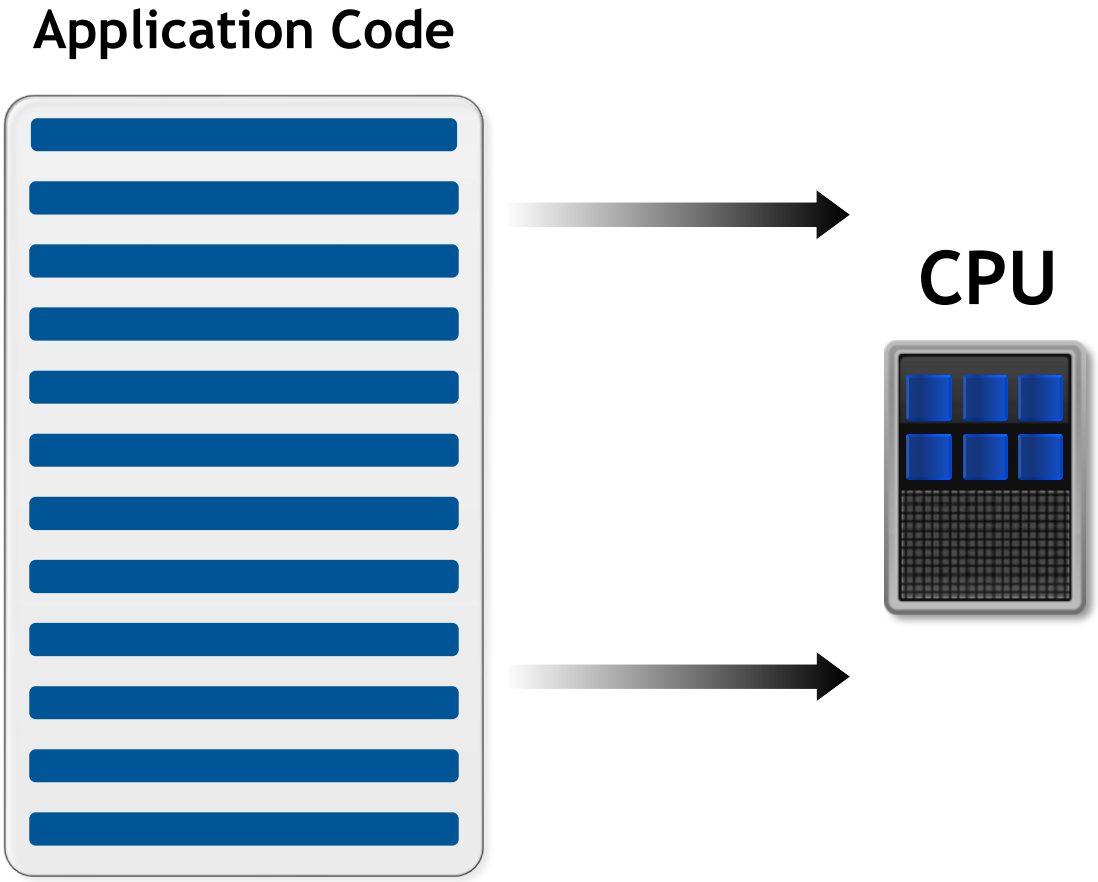
GPUs require massive number of threads to hide latency.

Summary -- from perspective of Memory Accesses



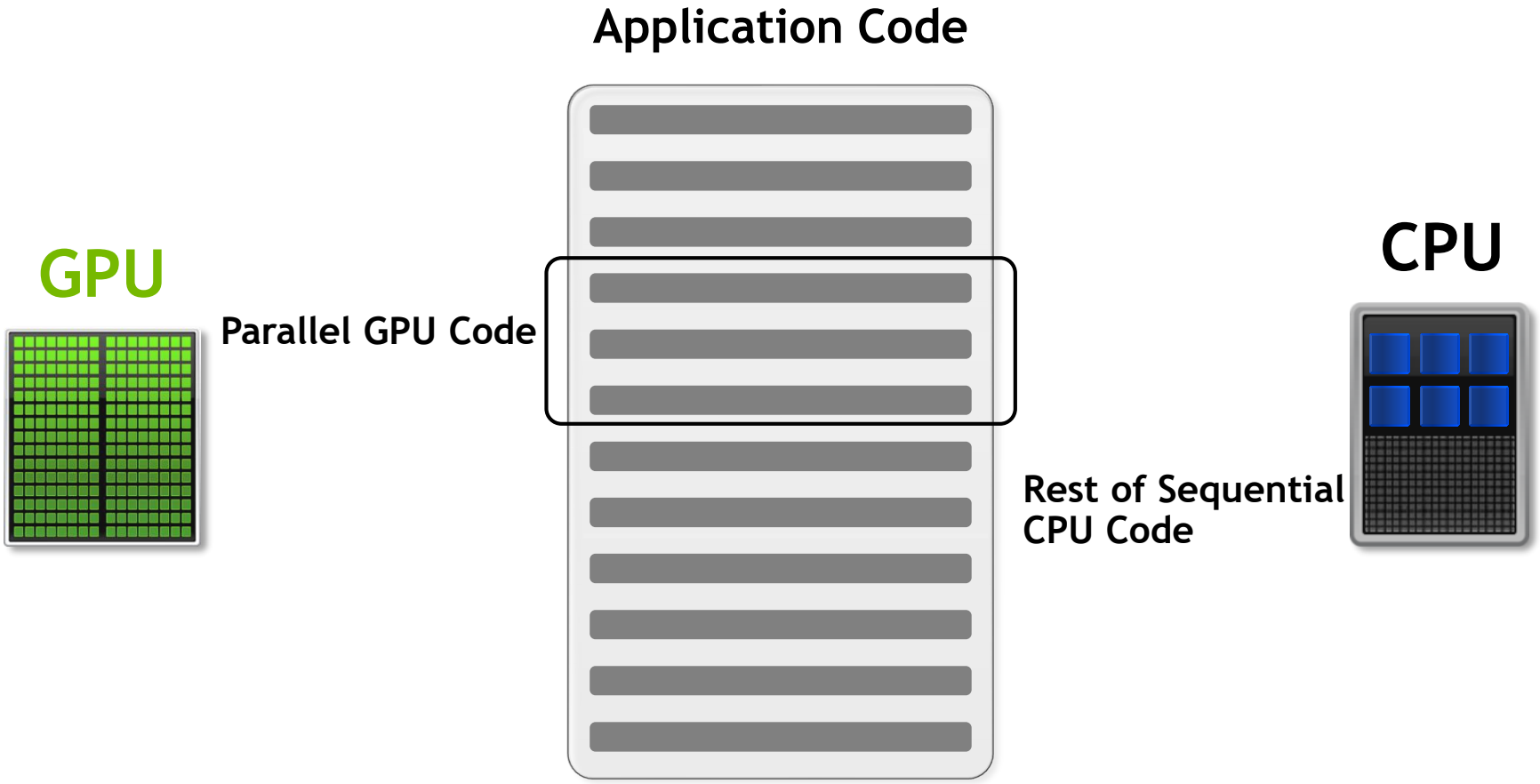
- **CPU** has host memory (**DDR4 SDRAM**)
  - Usually **tens to hundreds GB** in a compute node
  - Around **76.8 GB/sec** (DDR4) memory access bandwidth
- **GPU** has its own device memory (**GDDR5 or HBM or HBM2**)
  - Usually **6, 12, 16 GB** in a GPU
  - HBM2 (**NVIDIA** Tesla V100 GPU) has up to **900 GB/sec** peak bandwidth
- **GPU** is connected to the host via **PCIe or NVLink** connection
  - Data needs to move between host memory and device memory (between CPU and GPU)

Program Execution on CPU



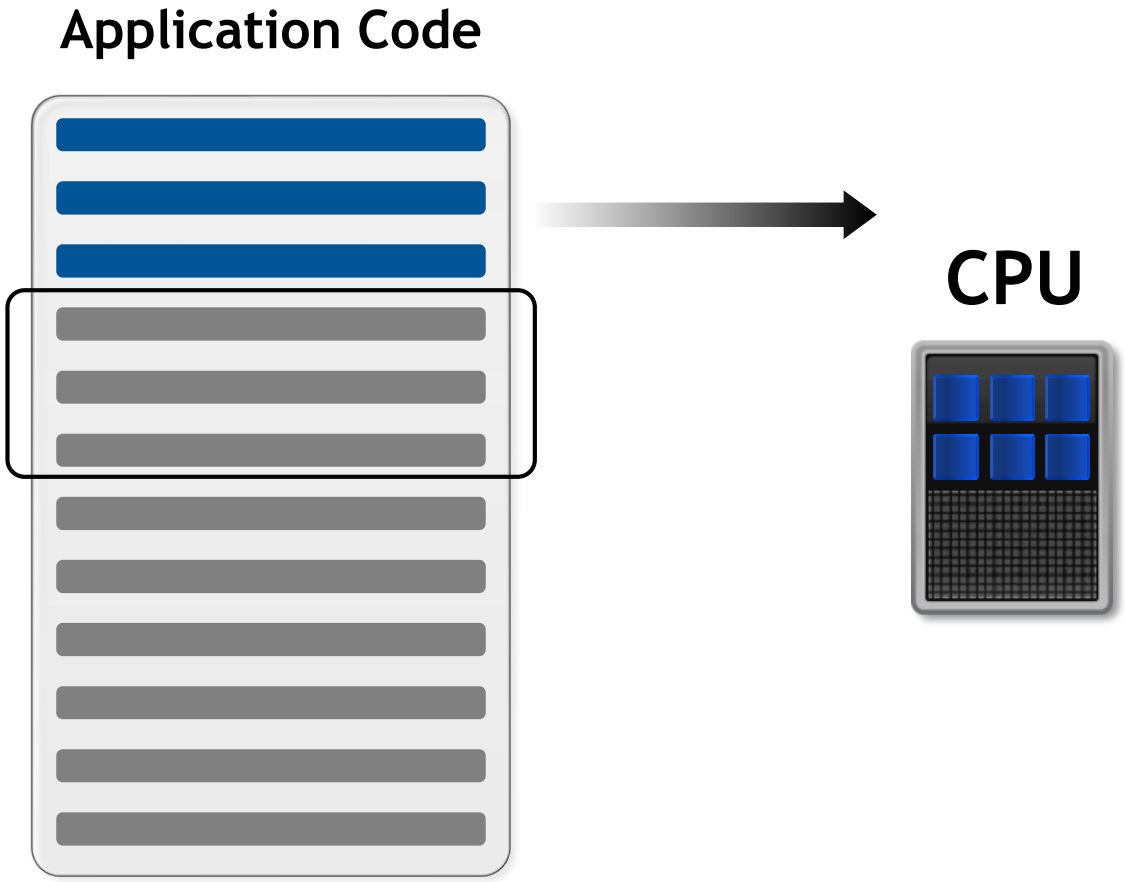
- **On CPU,** an application process will
  - Execute all instructions of the program
  - Read and write data from the main memory (via memory hierarchy)
  - Can make system calls and switch between the user mode and the kernel mode

Rewrite Program for GPU



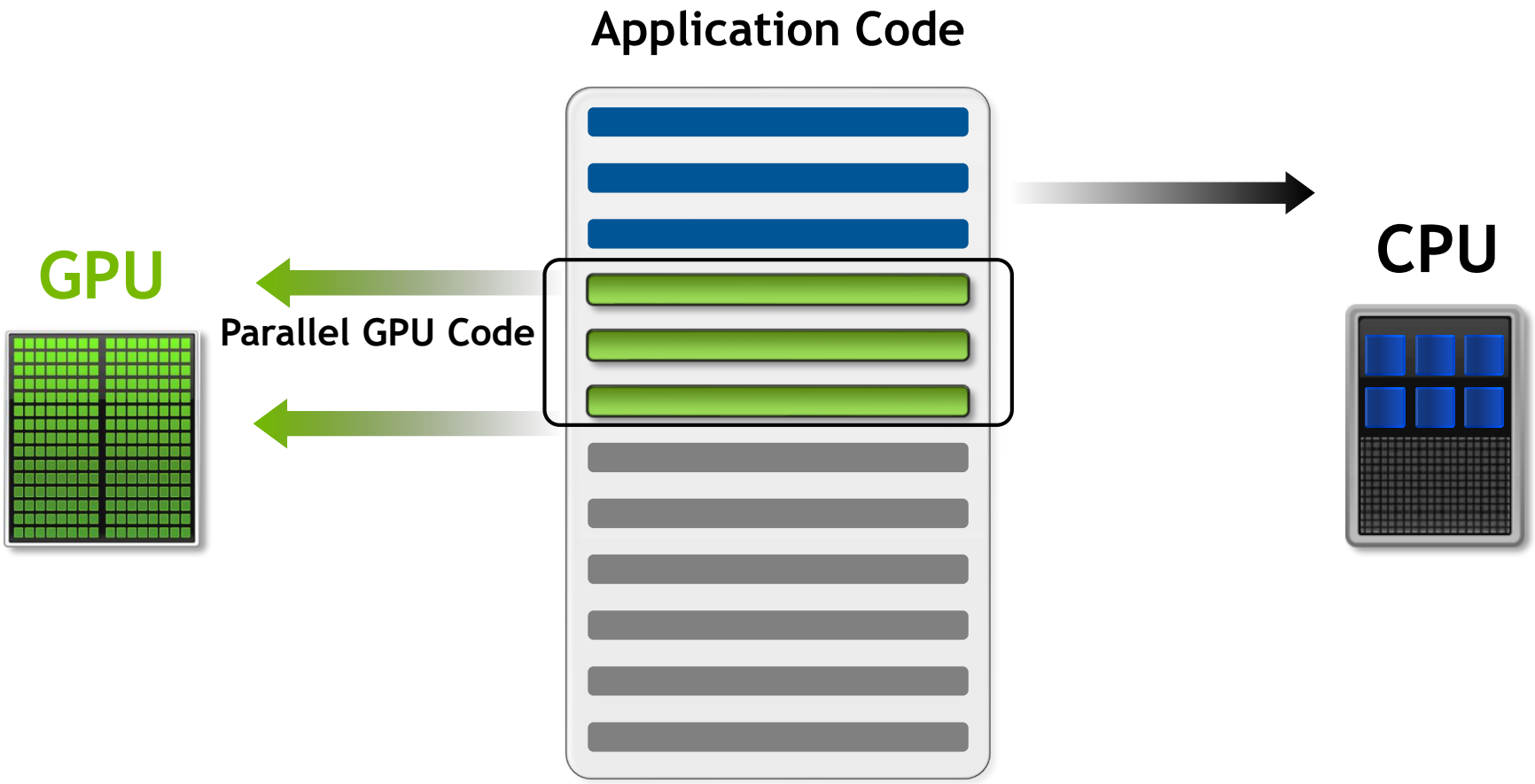
- The program needs to
  - Divide into **GPU part (for parallel execution)** + **CPU part (for sequential execution)**
  - Move input data into GPU device memory before GPU execution (**H2D**)
  - Move results back to host memory after GPU execution (**D2H**)

Program Execution on GPU



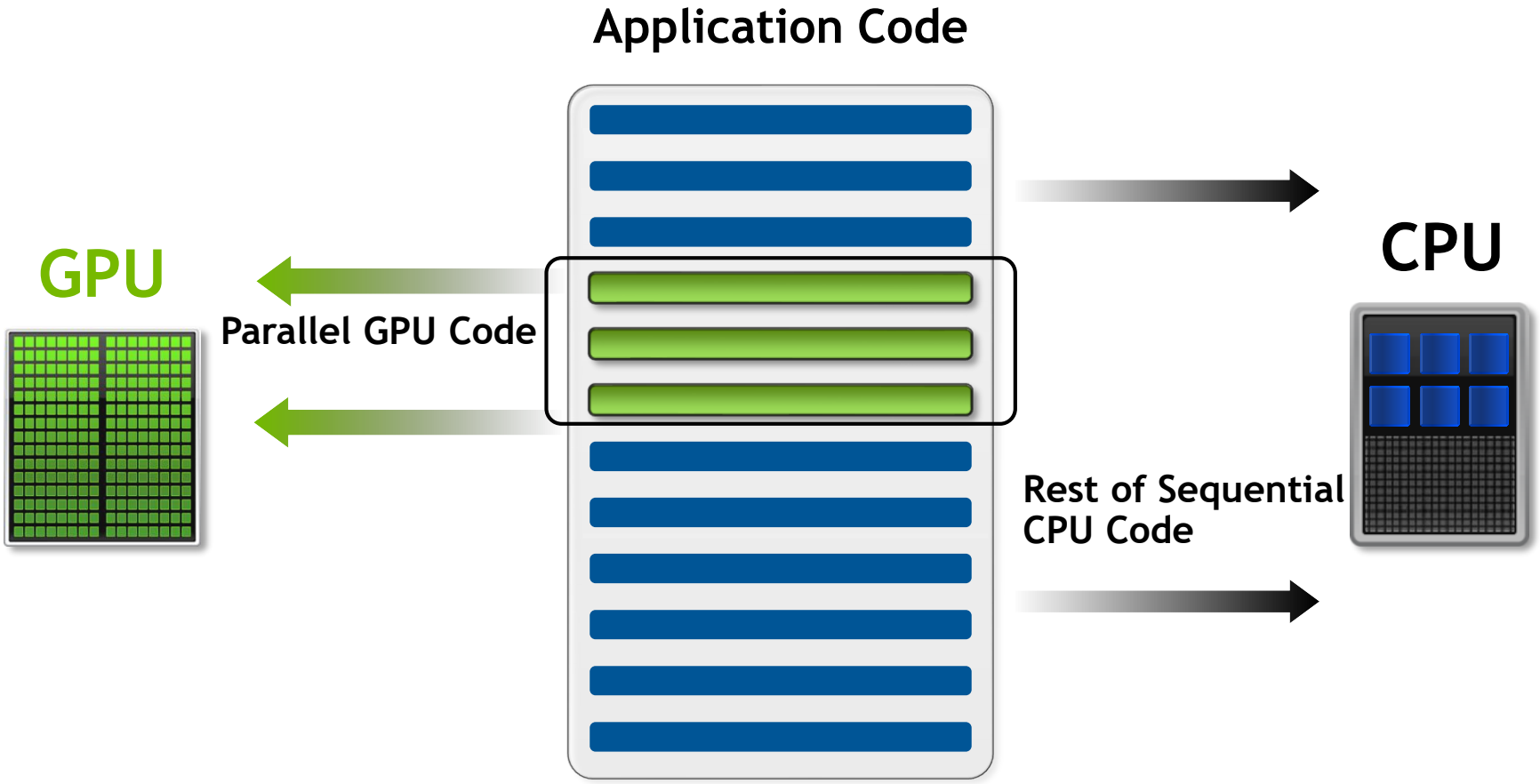
- An application will be executed in a **hybrid** mode on **CPU + GPU**
  - An application process is started on CPU

Program Execution on GPU



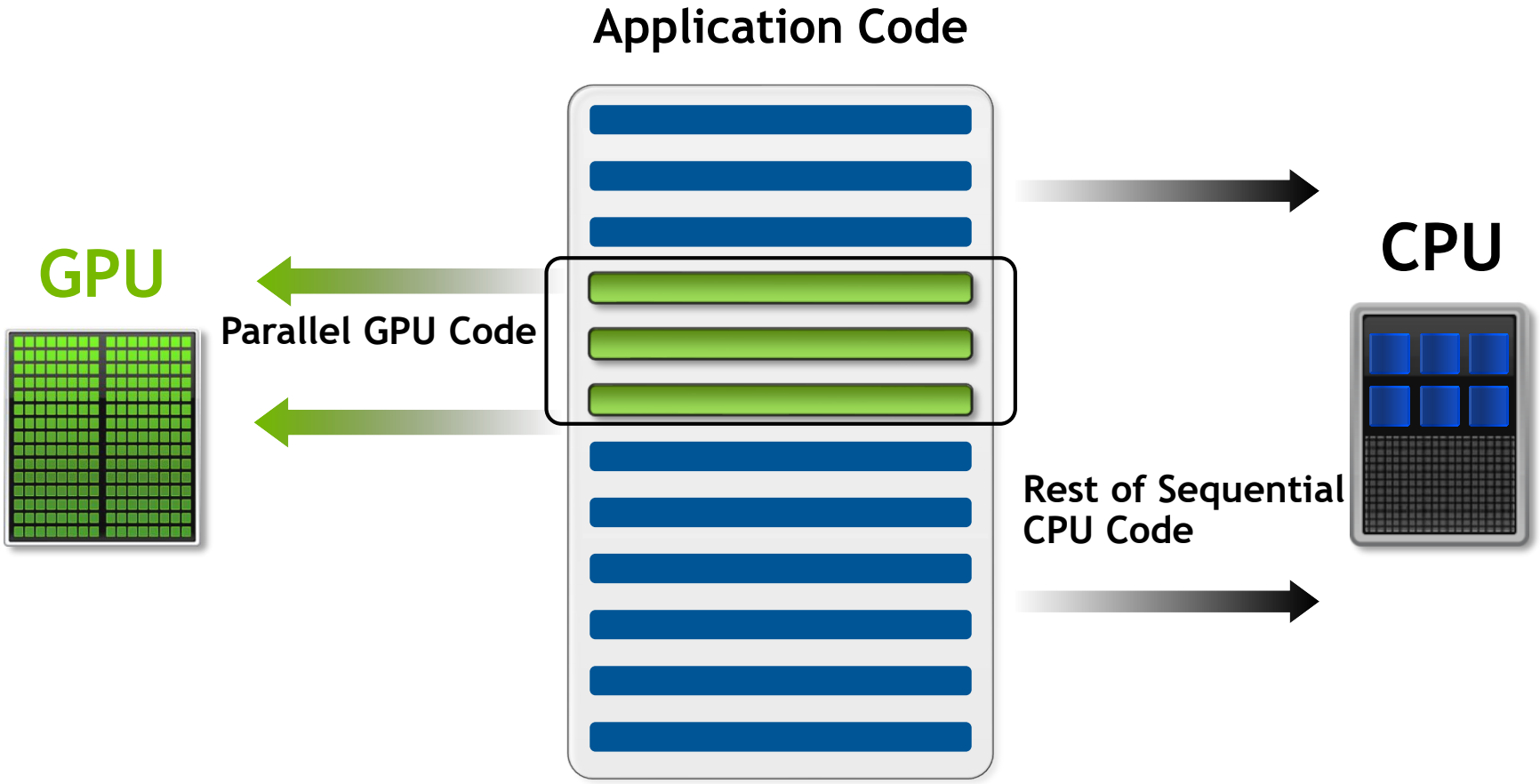
- An application will be executed in a **hybrid** mode on **CPU + GPU**
  - An application process is started on CPU
  - Launch GPU kernels to run parallel GPU code with multiple threads

Program Execution on GPU



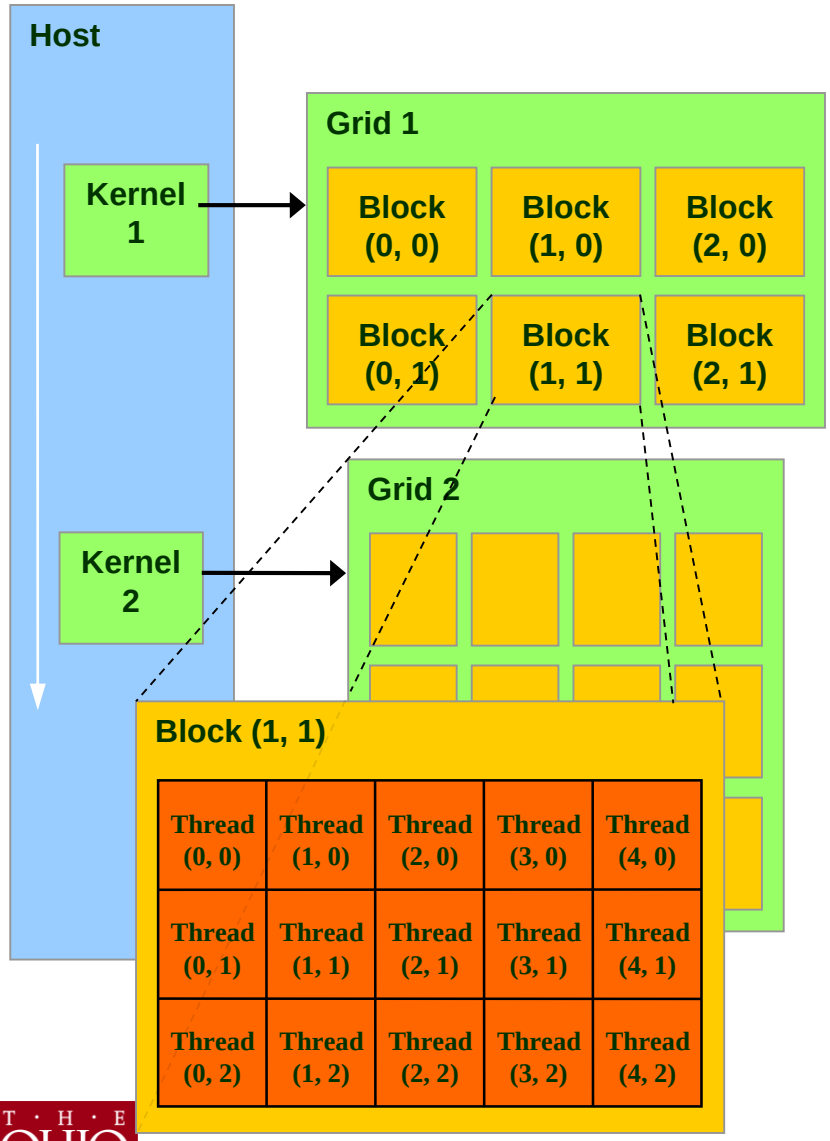
- An application will be executed in a **hybrid** mode on **CPU + GPU**
  - An application process is started on CPU
  - Launch GPU kernels to run parallel GPU code with multiple threads
  - Return to CPU to execute sequential CPU code

Program Execution on GPU



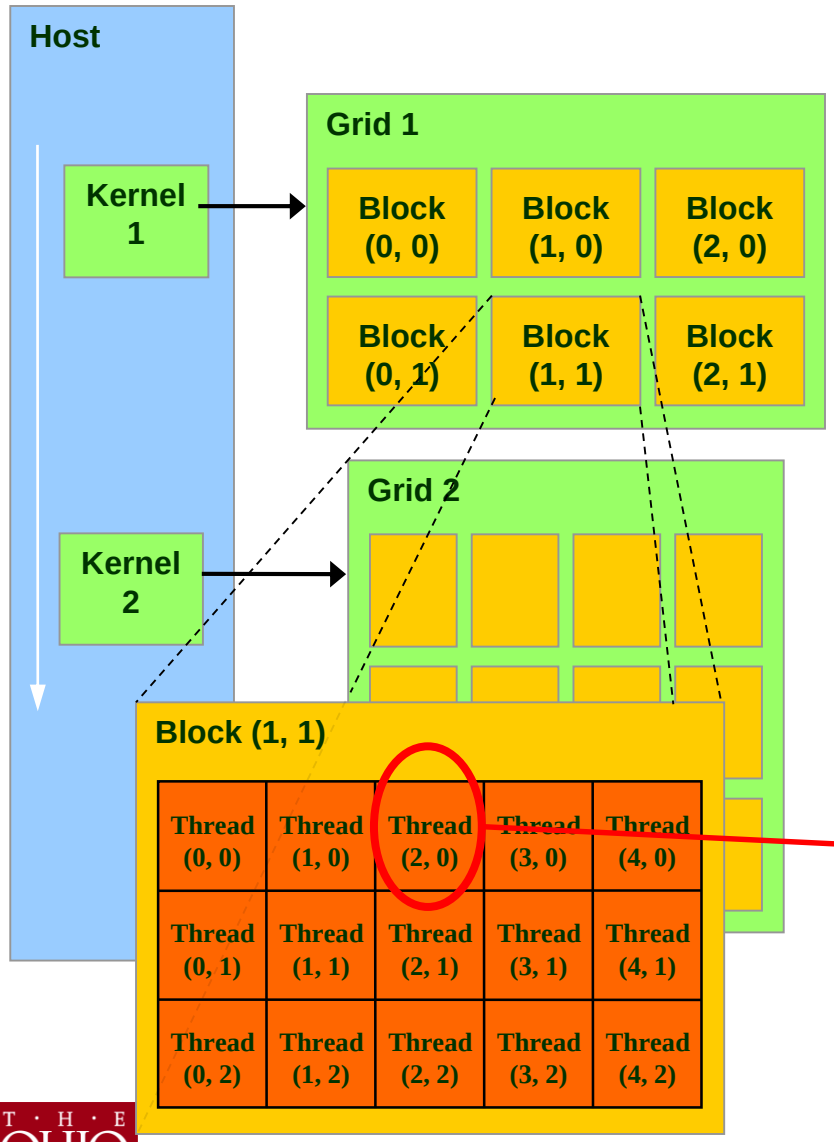
- An application will be executed in a **hybrid** mode on **CPU + GPU**
  - GPU code can access GPU memory space, including device memory, L1/L2 caches, shared memory, register files, and others
  - GPU code cannot access resources on CPU (host memory) or make system calls

CUDA Programming Concepts



- A **GPU kernel** is a function that can be executed on GPU
  - In a **Grid-Block-Thread** layout
  - Blocks of a grid and threads of a block can be arranged in 1, 2, or 3 dimensions

CUDA Programming Concepts



- **Kernel 1** is a two-dimensional kernel, having
  - 1 Grid
  - 6 thread blocks (2 * 3) in the grid
  - 15 threads (3 * 5) in each block
- Each thread can count its global thread IDs

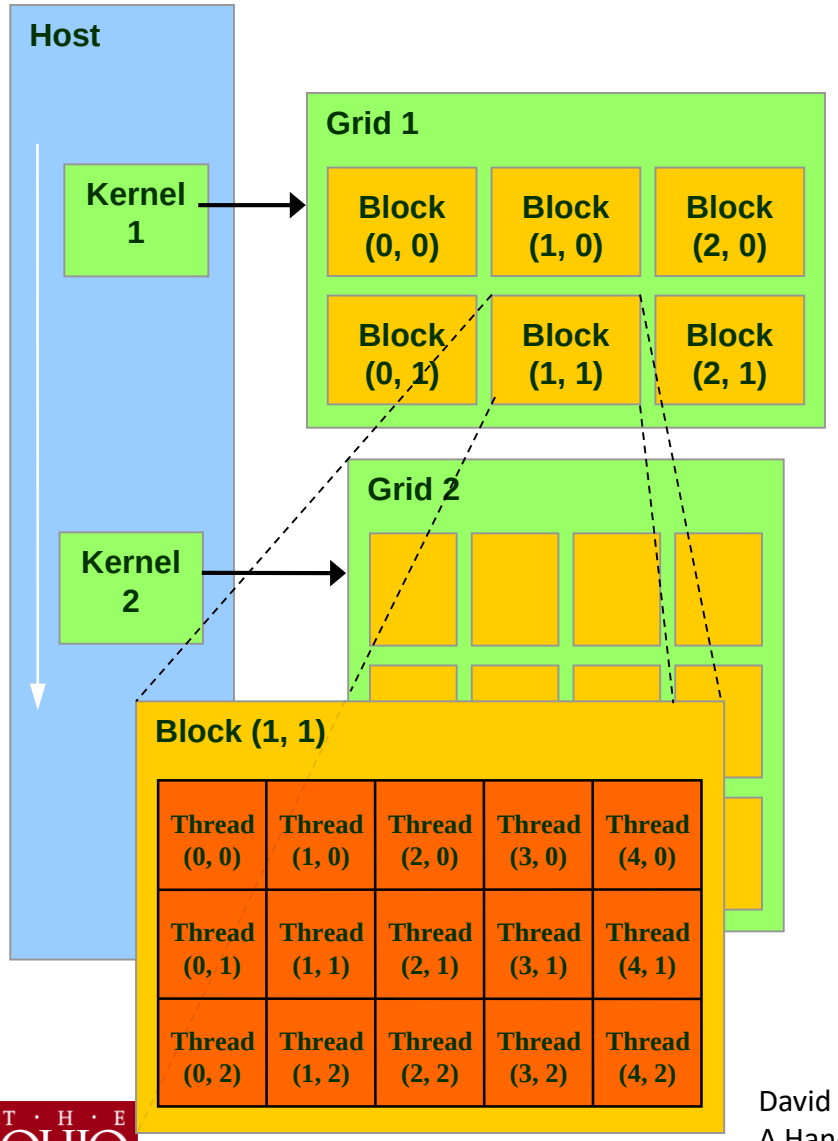
```
/* blockDim.x: how many threads in x dimension of a block*/
int idx = blockDim.x * blockIdx.x +
          threadIdx.x
int idy = blockDim.y * blockIdx.y +
          threadIdx.y
```

Thread (2, 0): blockDim.x is 5, blockIdx.x is 1, threadIdx.x is 2

idx is 5 * 1 + 2 = 7

idy is 3 * 1 + 0 = 3

CUDA Programming Concepts



- A **GPU kernel** is a function that can be executed on GPU
  - In a **Grid-Block-Thread** layout
  - Blocks of a grid and threads of a block can be arranged in 1, 2, or 3 dimensions
  - All threads share GPU device memory
- A **block** of threads are scheduled on the same SM (by **Thread Block Scheduler**)
  - Share L1 cache and shared memory in a SM
  - Cooperate with others by synchronization functions (CUDA API)
- A **warp** is a group of 32 threads
  - 32 threads in a warp are scheduled to execute code together (by **Warp Scheduler**)
  - A thread block consists of multiple warps

David B. Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach, 3rd Edition", Morgan Kaufmann, December 7, 2016

CUDA Example: *Vector Add (C = A + B)*

CPU code for *vector add*

```
void add (float *A, float *B, float *C) {
    for (i = 0; i < N; i++)
        C[i] = A[i] + B[i];
}

int main(void) {
    /* allocate memory for A, B, C; work on A and B */
    add(A, B, C);
    /* continue working on C; free A, B, C */
}
```

CUDA Example: *Vector Add (C = A + B)*

```
/* Main function, executed on host (CPU) */
int main(void) {

    /* 1. Allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
```

CUDA Example: *Vector Add (C = A + B)*

```
/* Main function, executed on host (CPU) */
int main(void) {
    /* 1. Allocate memory on GPU */
    float *d_A = NULL, *d_B = NULL, *d_C = NULL;

    if (cudaMalloc((void **)&d_A, size) != cudaSuccess)
    exit(EXIT_FAILURE);

    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);
    ……
    return(0);
}
```

CUDA Example: *Vector Add (C = A + B)*

```
/* Main function, executed on host (CPU) */
int main(void) {
      /* 1. Allocate memory on GPU */

      /* 2. Copy data from Host to GPU */
      cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
      cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
      ……

      return(0);
}
```

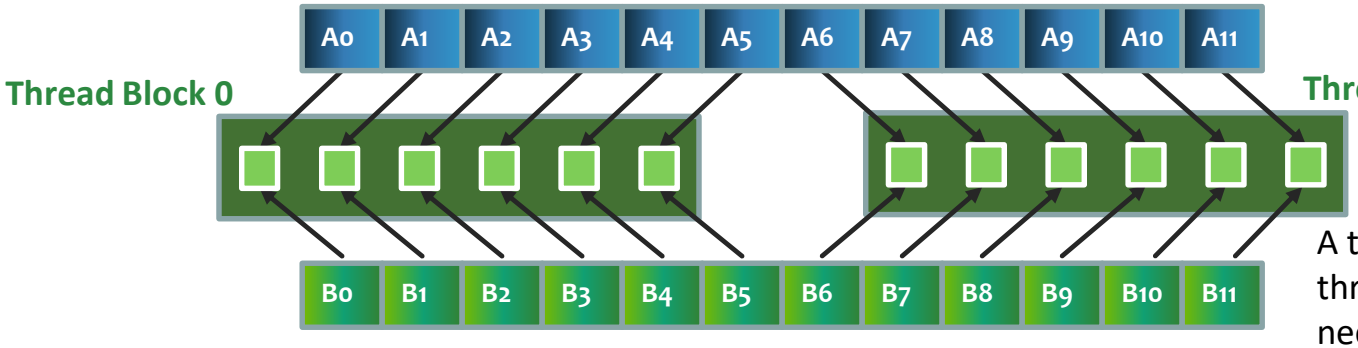
CUDA Example: *Vector Add (C = A + B)*

```
/* Main function, executed on host (CPU) */
int main(void) {

    /* 3. Execute GPU kernel */
    /* Calculate number of blocks and threads */
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock
                        - 1) / threadsPerBlock;
    /* Launch GPU kernel */
    add<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B,
                        d_C, numElements);
    /* Wait for all the threads to complete */
    cudaDeviceSynchronize();
    ……
    return(0);
}
```

CUDA Example: *Vector Add (C = A + B)*

```
/* Main function, executed on host (CPU) */
int main(void) {

    /* 4. Copy data from GPU back to Host */
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    /* 5. Free GPU memory */
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return(0);
}
```

CUDA Example: *Vector Add (C = A + B)*



A thread block has 6 threads. Two blocks are needed in this case.

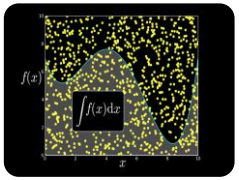
```
/* GPU Kernel */
__global__ void add(float *A,
                    float *B,
                    float *C,
                    int numElements) {
    /* Calculate the position in A, B, C*/
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    /* Add A[i] and B[i] to C[i] */
    if (i < numElements) C[i] = A[i] + B[i];
}
```

Let's Recall *Vector Add* Example

```
/* Main function, executed on host (CPU) */
int main(void) {

    /* 1. Allocate memory on GPU */

    /* 2. Copy data from Host to GPU */

    /* 3. Execute GPU kernel */

    /* 4. Copy data from GPU back to Host */

    /* 5. Free GPU memory */

    return(0);
}
```

Widely Used Libraries on GPU

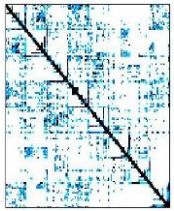
NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA NPP

OpenCV



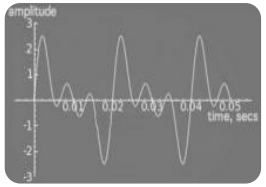
NVIDIA cuSPARSE

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA



NVIDIA cuFFT

GPU Libraries -- Thrust

- https://developer.nvidia.com/thrust

Thrust is a powerful library of parallel algorithms and data structures;

C++ developers can write just a few lines of code to perform GPU-accelerated sort, scan, transform, and reduction operations orders of magnitude faster than the latest multi-core CPUs.

Reduction

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <cstdlib>

int main(void)
{
  // generate random data on the host
  thrust::host_vector<int> h_vec(100);
  thrust::generate(h_vec.begin(), h_vec.end(), rand);

  // transfer to device and compute sum
  thrust::device_vector<int> d_vec = h_vec;
  int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>());
  return 0;
}
```

GPU Libraries -- cuBLAS

- https://developer.nvidia.com/cublas

cuBLAS is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS).

Complete support for all 152 standard BLAS routines, for single, double, complex, and double complex data types.

GEMM: C = αAB + βC

$$C(i,j) = \alpha * \sum_{k=0}^{n-1} A(i,k) * B(k,j) + \beta * C(i,j)$$

```
/* General Matrix Multiply */
static void simple_dgemm(int n, double alpha, double *A,
                         double *B, double beta, double *C) {
   int i, j, k;
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++){
         double prod = 0;
         for (k = 0; k < n; k++)
             prod += A[k * n + i] * B[j * n + k];
         C[j * n + i] = alpha * prod + beta * C[j * n + i];
      }
   }
}
```

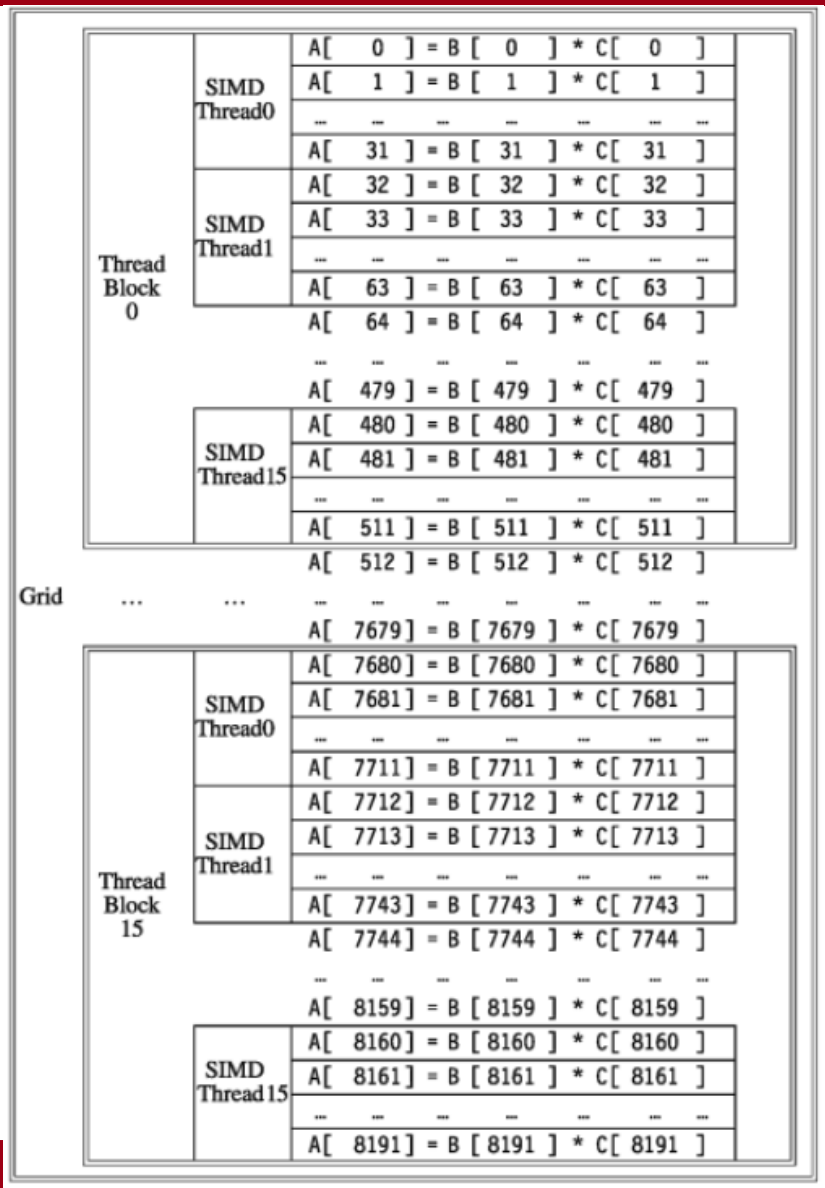
Use **cuBLAS**: C = αAB + βC

```
/* dgemm from BLAS library */
extern "C"{
extern void dgemm_(char *, char * ,
                   int *, int *, int *,
                   double *, double *, int *,
                   double *, int *,
                   double *, double *, int *); };

int main(int argc, char **argv) {
   . . .
   /* call gemm from BLAS library */
   dgemm_("N","N", &N, &N, &N, &alpha, h_A, &N, h_B, &N, &beta, h_C_blas, &N);
   . . .
}
```

Use **cuBLAS**: **C = αAB + βC**

```
/* Main */
int main(int argc, char **argv) {
     /* 0. Initialize CUBLAS */
     cublasCreate(&handle);

     /* 1. allocate memory on GPU */
     cudaMalloc((void **)&d_A, n2 * sizeof(d_A[0]));

     /* 2. Copy data from Host to GPU */
     status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);

     /* 3. Execute GPU kernel */
     cublasDgemm( handle,
           CUBLAS_OP_N, CUBLAS_OP_N, N, N, N, &alpha, d_A, N, d_B, N, &beta, d_C, N );

     /* 4. Copy data from GPU back to Host */
     cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);

     /* 5. Free GPU memory */
     cudaFree(d_A);
}
```

GEMM Kernel on GPU

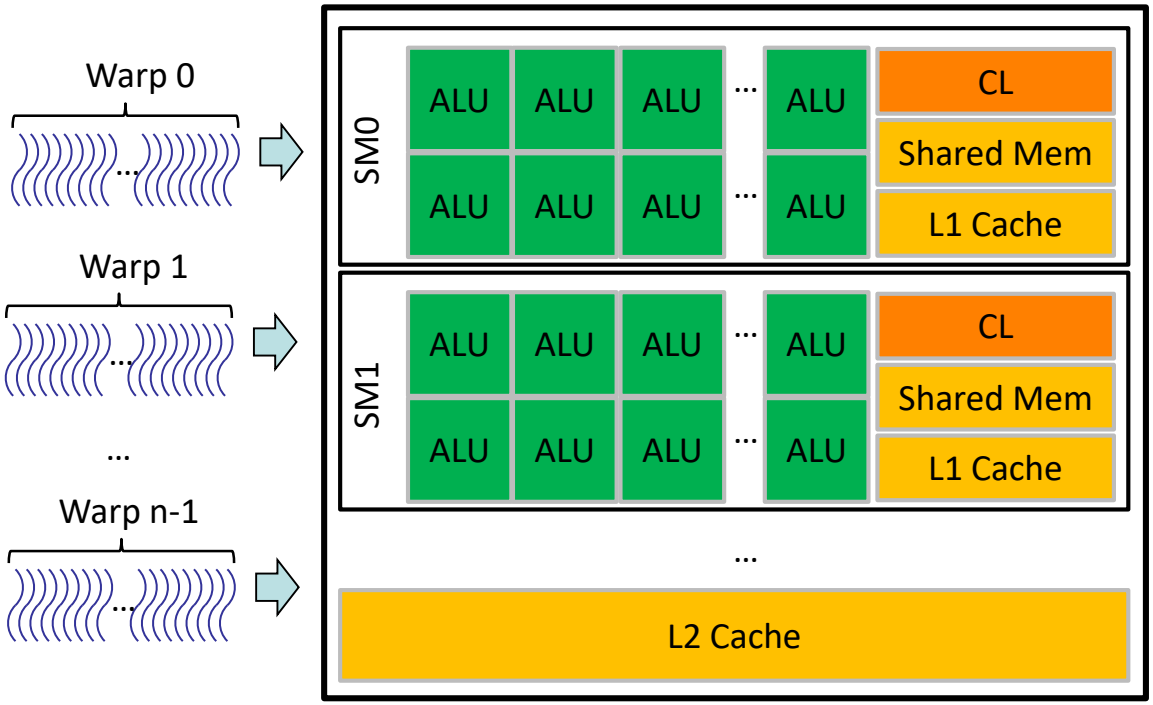


- The GEMM kernel includes
  - 1 Grid
  - 16 thread Blocks in the grid
  - 16 Warps (SIMD Threads) in each block
  - 32 threads in each warp
- **1 * 16 * 16 * 32 = 8192** parallel threads

John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach 6th Edition", Morgan Kaufmann , 2017

Performance on GPU

- **Shared control logics**
  - All threads in a **warp** (32 threads) need to execute the same instruction at the same time
  - All threads in a warp require to access consecutive memory addresses for efficient memory load and store

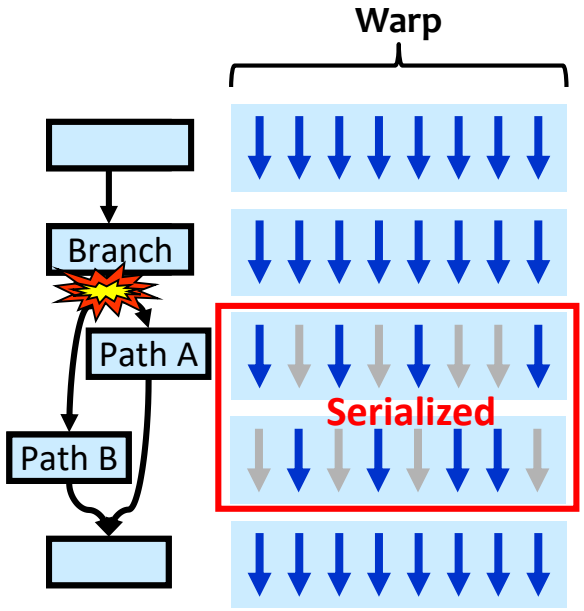


Performance on GPU is sensitive to memory accesses and control flows.

To Avoid: Branch Divergence

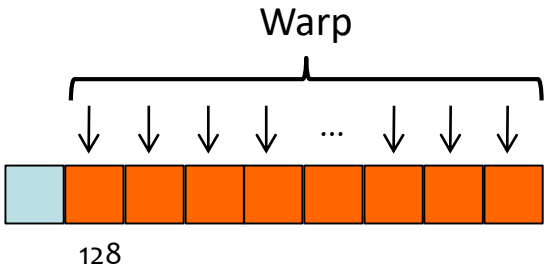
- Branch divergence will waste computational resources
- When a GPU kernel has a branch
  - Some threads will go into Path A, and the rest will go into Path B
  - All threads will first execute Path A, and then execute Path B
  - Hardware supported execution mask will control the write back of results
  - The execution of Paths A and B is serialized

```
Kernel(…){
    …
    if(a[tid]){
        //Path A
        …
    }else{
        //Path B
        …
    }
}
```

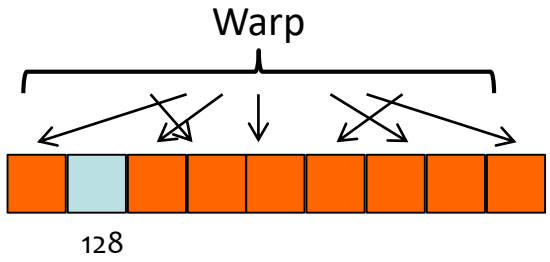


To Avoid: Uncoalesced Memory Accesses

- Coalesced memory accesses
  - If a warp of threads access consecutive memory addresses in a memory segment, data can be loaded in a single memory transaction
- Un-coalesced memory accesses
  - If a warp of threads access different memory addresses in a memory segment, data will be loaded in multiple memory transactions
- Uncoalesced memory accesses will lower memory bandwidth

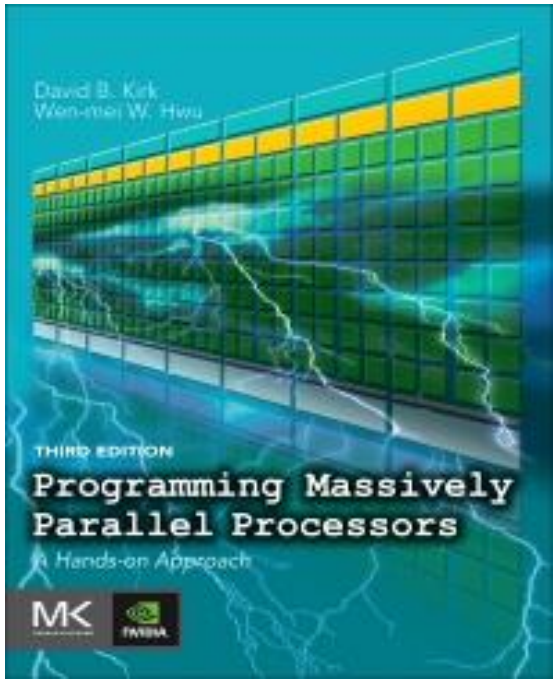


Coalesced Memory Accesses
Consecutive accesses on a memory segment
Loaded by a single 128-byte transaction

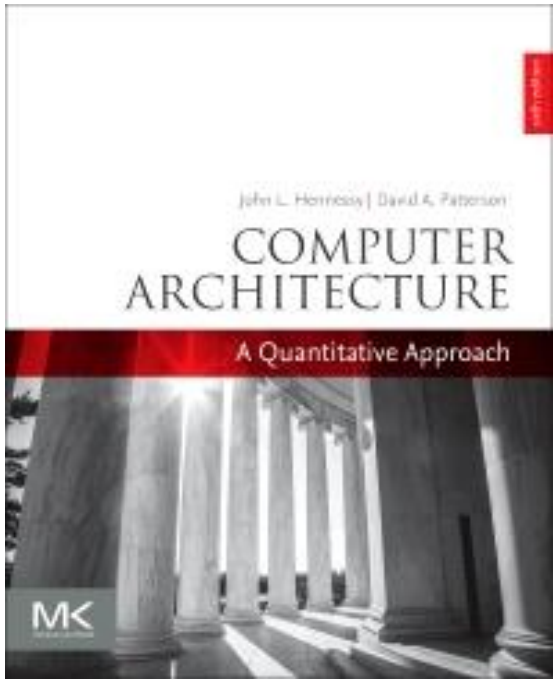


Uncoalesced Memory Accesses
Random accesses across N segments
Loaded by N 32-byte transaction

Further Reading



David B. Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach, 3rd Edition", Morgan Kaufmann, December 7, 2016



John L. Hennessy, David A. Patterson, "Computer Architecture: A Quantitative Approach 6th Edition", Morgan Kaufmann , 2017