GPU SM ARCHITECTURE & PROGRAMMING MODEL

COMP4300/8300 PARALLEL SYSTEMS

PROF. JOHN TAYLOR

MAY 2024

Australian National University

Logistics

- Attendance to the Lab sessions is highly encouraged. Most of the practical aspects of the programming models are covered in the Labs.

NVIDIA V100 GPU AT NCI -2D MATRIX SUM

Device	Version	Grid	Block	Time	Speedup
CPU	matrix-add-cpu	Nx, Ny = 32768	N/a	31,366 ms	1
CPU	matrix-add-openmp-avx	Nx, Ny = 32768	N/A	3302 ms	9.5
CPU	matrix-add-openmp-gcc	Nx, Ny = 32768	N/A	550 ms	57
GPU	matrix-add-gpu	1024 x 1024	32 x 32	19.23 ms	1631
GPU	matrix-add-gpu	1024 x 2048	32 x 16	18.40 ms	**1704**
GPU	matrix-add-gpu	2048 x 1024	16 x 32	21.38 ms	1467
GPU	matrix-add-gpu	2048 x 2048	16 x 16	18.73 ms	1674

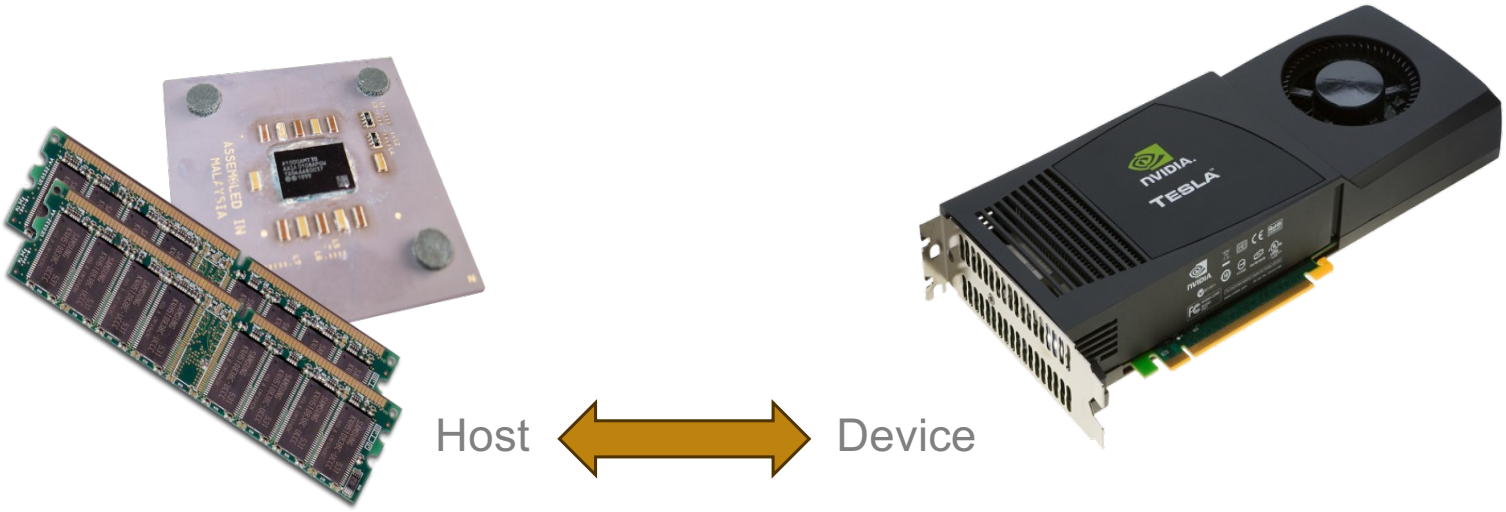
Heterogeneous Computing

Reference Material

- NVIDIA's CUDA C++ Best Practices Guide, https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/
- Nvidia H100 TensorCore GPU Architecture https://resources.nvidia.com/en-us-tensor-core
- Jia, Z., Maggioni, M., Staiger, B., & Scarpazza, D. P. (2018). *Dissecting the NVIDIA volta GPU architecture via microbenchmarking*. arXiv preprint arXiv:1804.06826.
- *Professional CUDA c programming*. Cheng, John, Max Grossman, and Ty McKercher. John Wiley & Sons, 2014.
- *CUDA by Example: An Introduction to General-Purpose GPU Programming*, Sanders, Jason, and Edward Kandrot, Addison-Wesley Professional, 2010.
- Tesla V100 Performance Optimization Guide, https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/ tesla-product-literature/v100-application-performance-guide.pdf

Heterogeneous Computing

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



Heterogeneous Computing

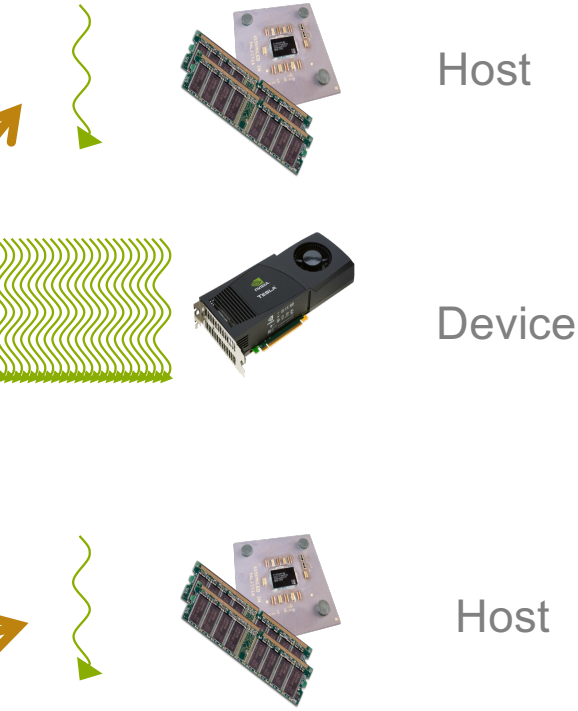
```
#include <iostream>
#include <algorithm>

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;
}
```

parallel fn

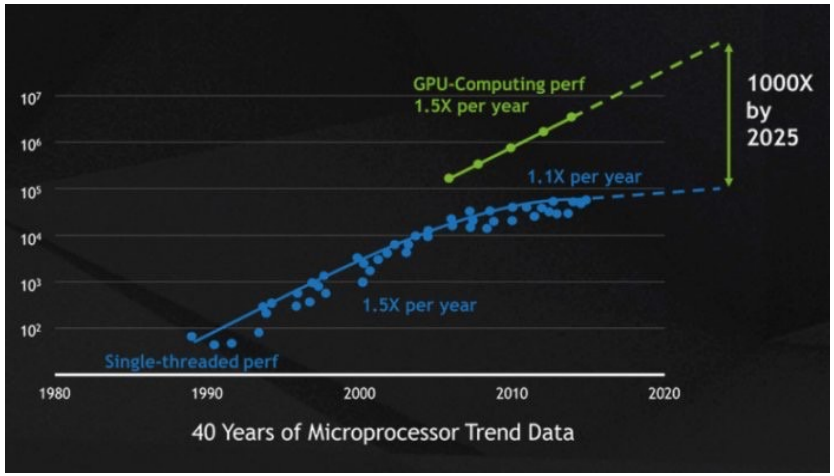
serial code

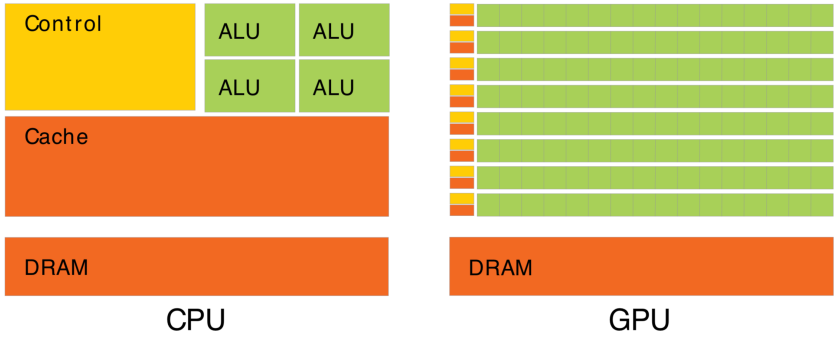
parallel code

serial code



The End of The Road for General-Purpose Processors

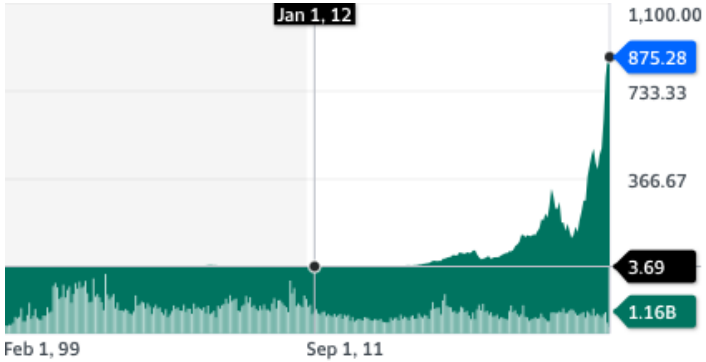




- End of Dennard scaling caused the end of the general-purpose processor era (both uniprocessor and multicore)
- Use of domain specific architectures (DSAs): programmable but designed for a class of problems with specific structures.
- GPUs are designed for data-parallel algorithms (especially linear algebra)
- More transistors are devoted to data processing rather than data caching and flow control
- Require domain specific programming model that makes it possible for the software to match the hardware (e.g. CUDA)
- Extracting performance requires the programmer to expose parallelism, to manage memory efficiently (e.g. caching), to tailor the algorithm to the hardware

TOP 500 List November 2023

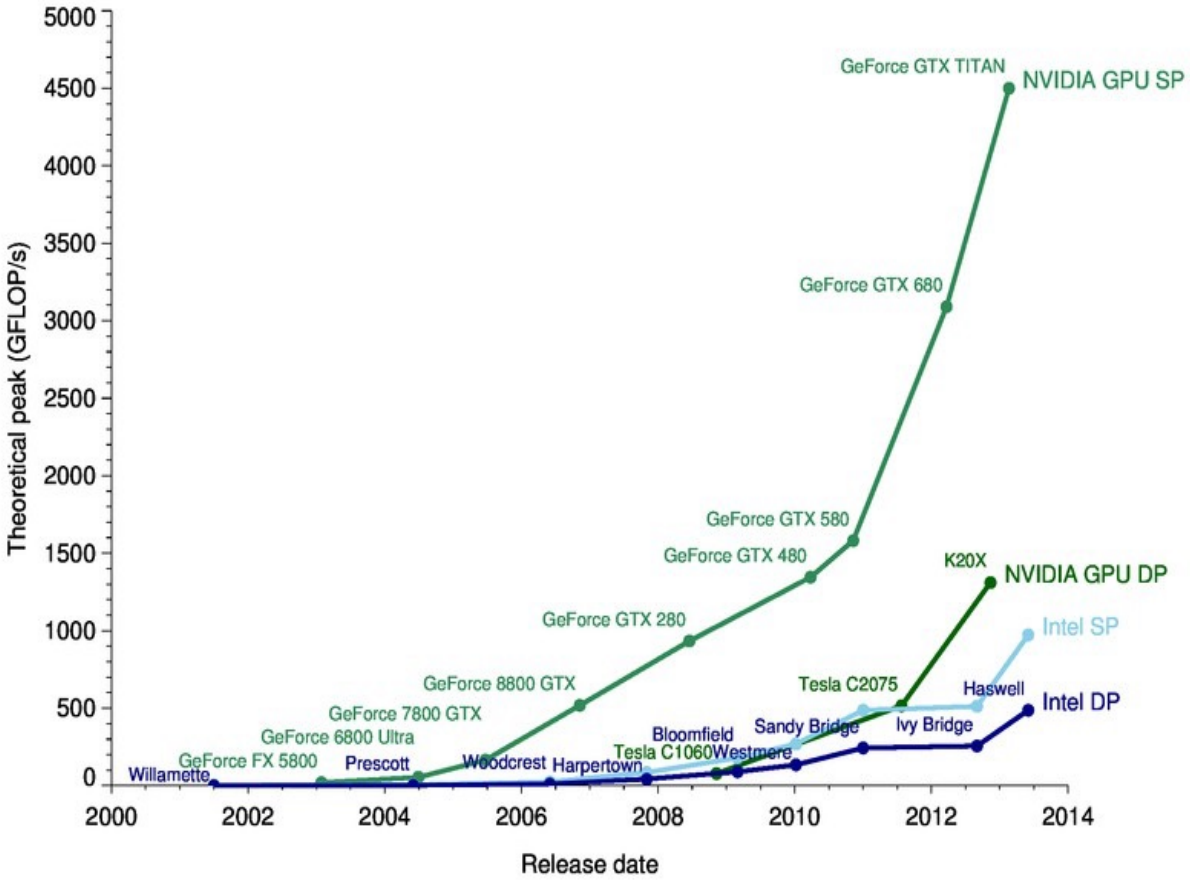
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	4,742,808	585.34	1,059.33	24,687
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Microsoft Azure United States	1,123,200	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107



TOP 500
The List.

https://www.top500.org/

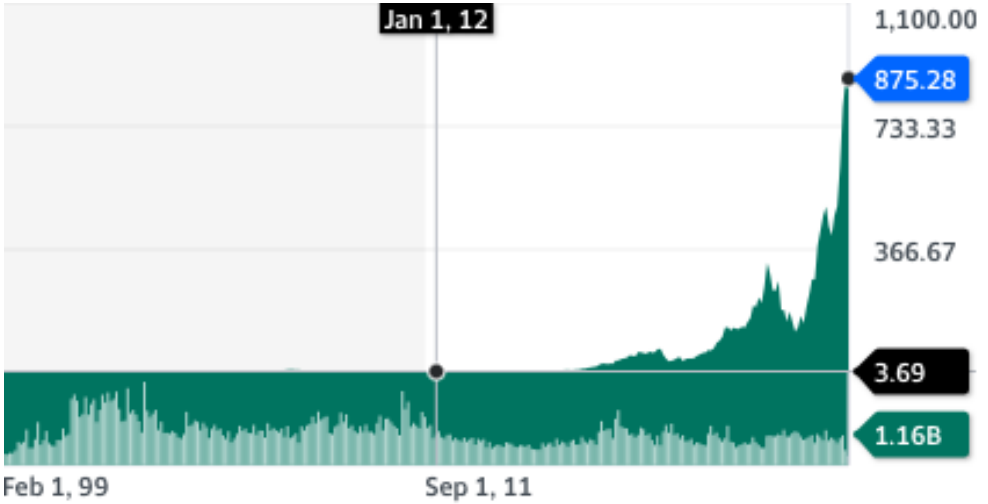
CPU versus GPU - FLOP rates

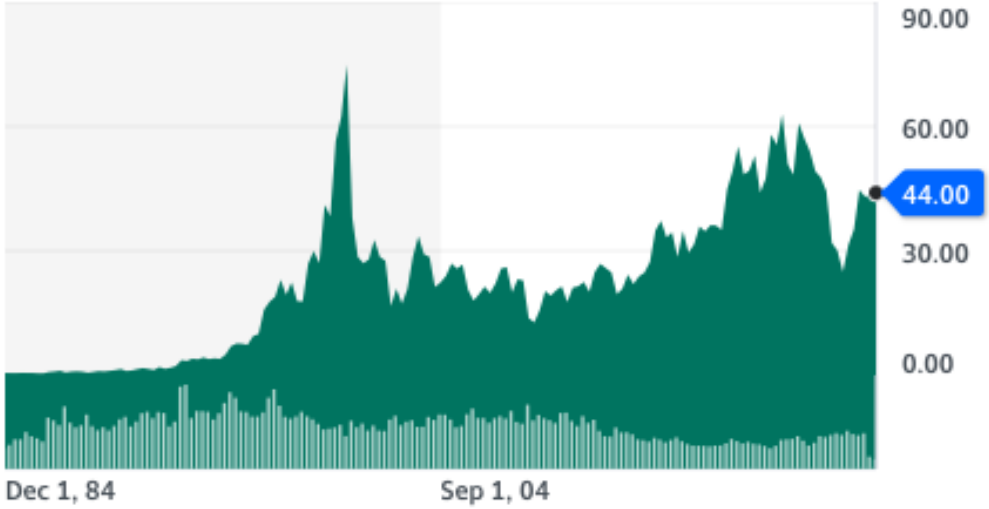


GPU FLOP Rates have been growing exponentially:-

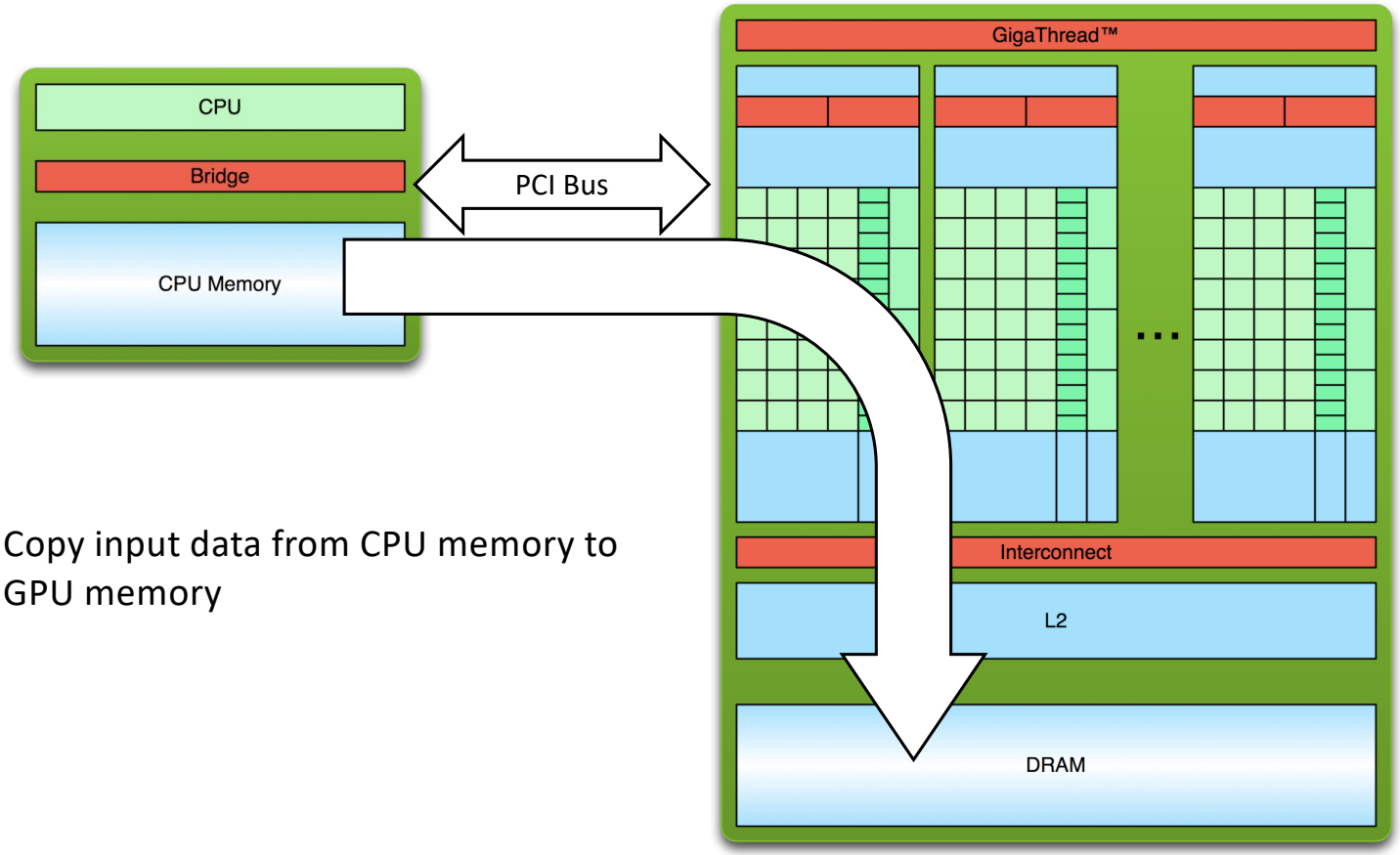
- 2010's GFLOP/s – see the graph opposite
- 2020's TFLOPS/s to PFLOP/s e.g. H100 GPU

Impact of Heterogenous Computing



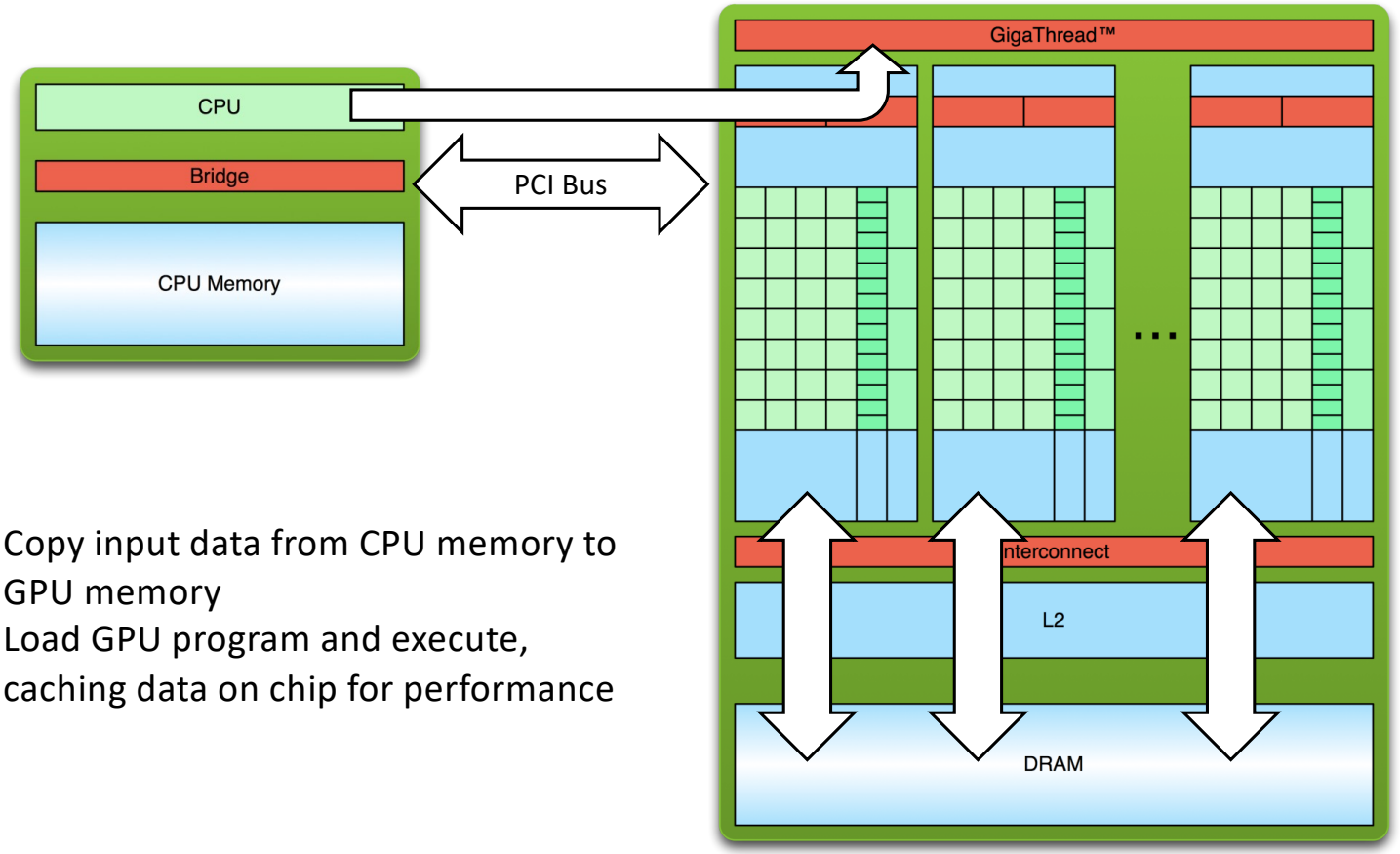


Simple Processing Flow



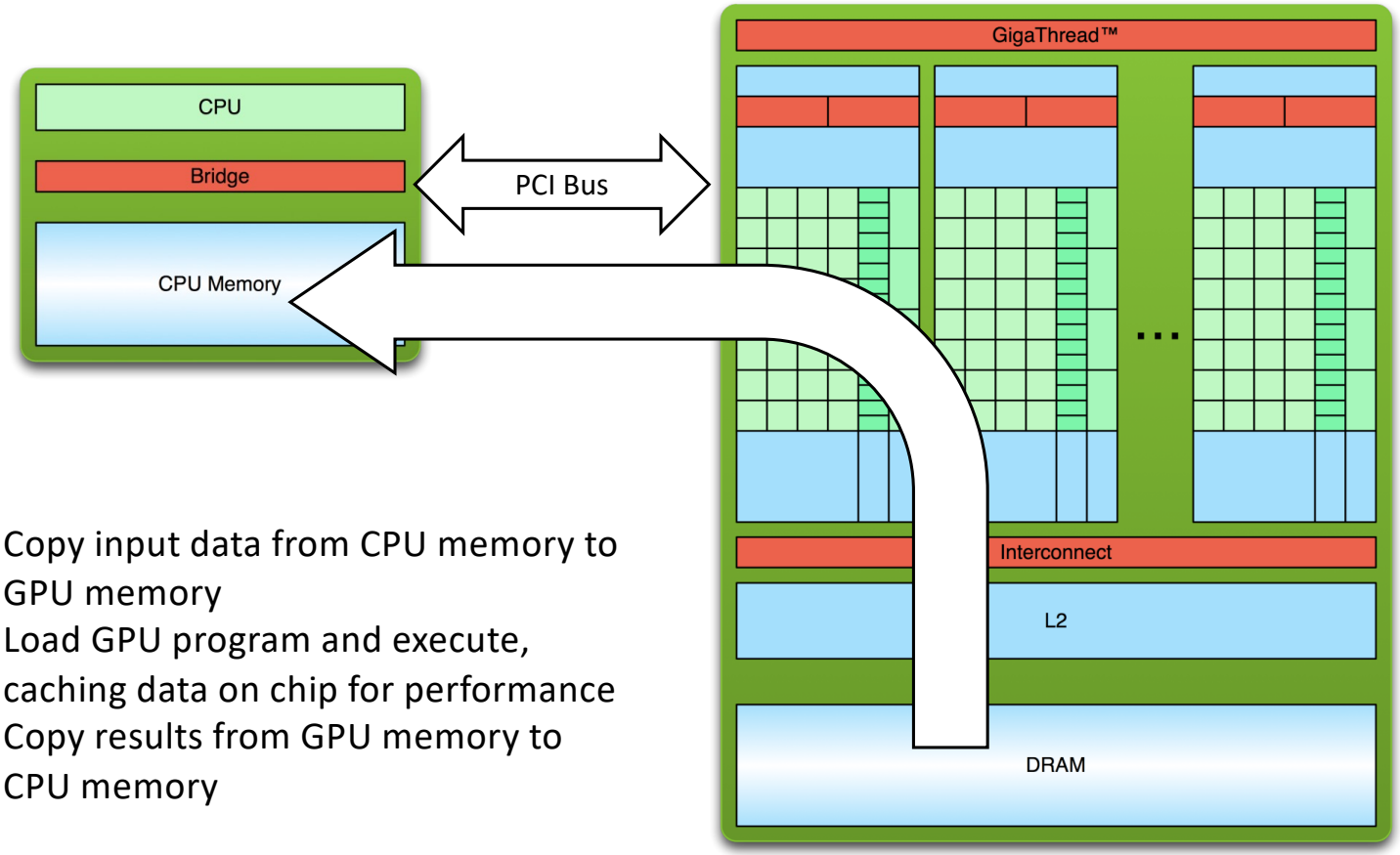
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



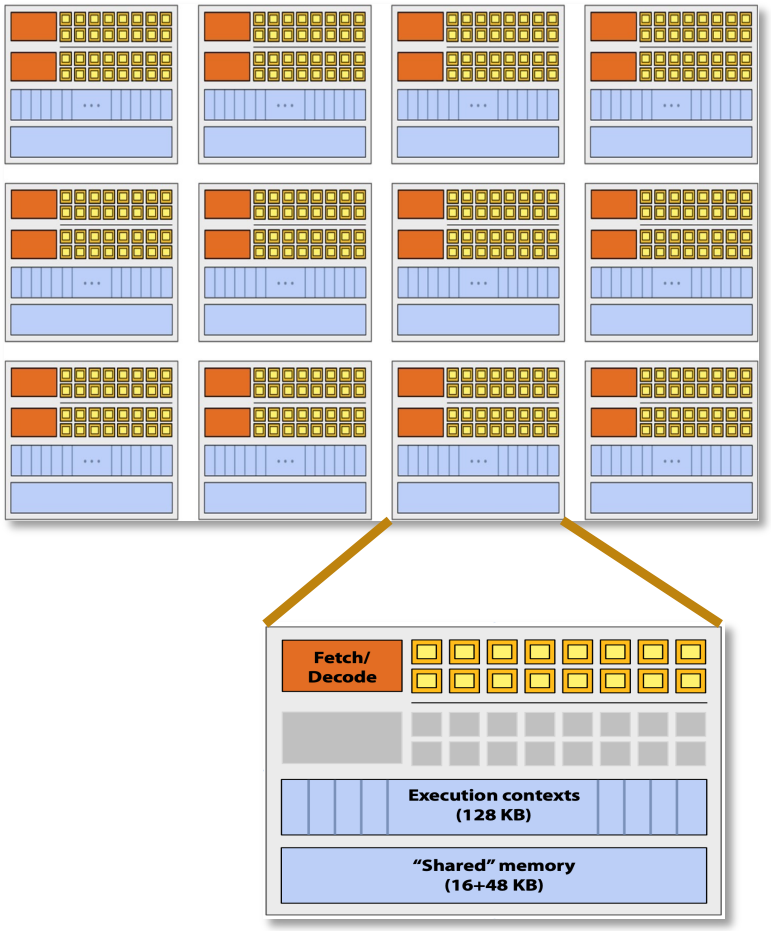
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

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

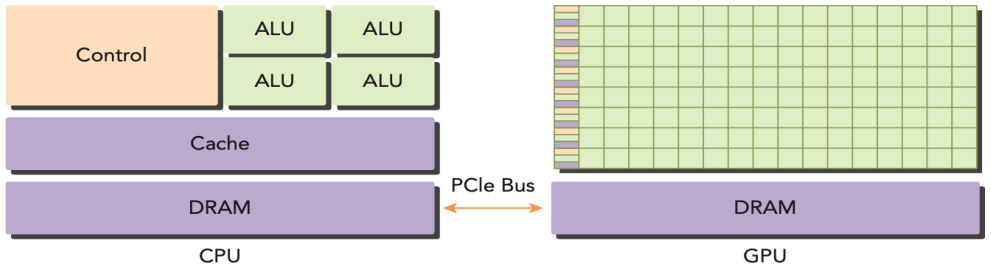
A Simplistic View of the GPU Architecture

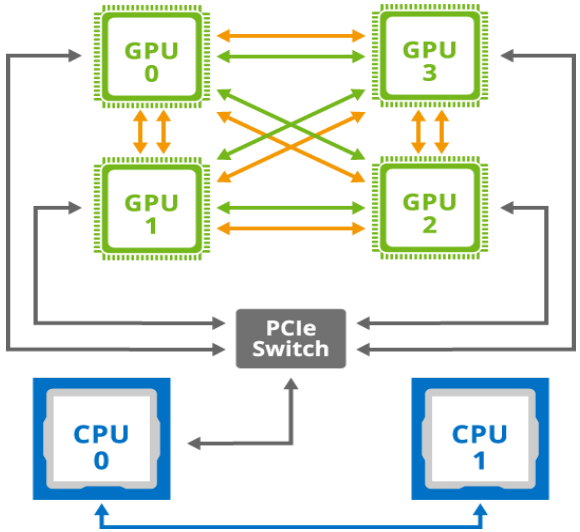


A scalable array of complex "cores" called Streaming Multiprocessors (SM)

- Each core has an array of functional units (e.g. ALUs) with SIMD execution
- Instructions operate in groups of 32 "SIMD" threads called warps
- On the NVIDIA H100 GPU up to 64 warps can be executed concurrently (interleaved) on a single SM
- Up to 132 SMs × 128 CUDA cores/SM = 16896 Cuda cores per device
- H100 includes Tensor cores + Transformer engine for training large language models
- This is why GPUs are called throughput-oriented architectures

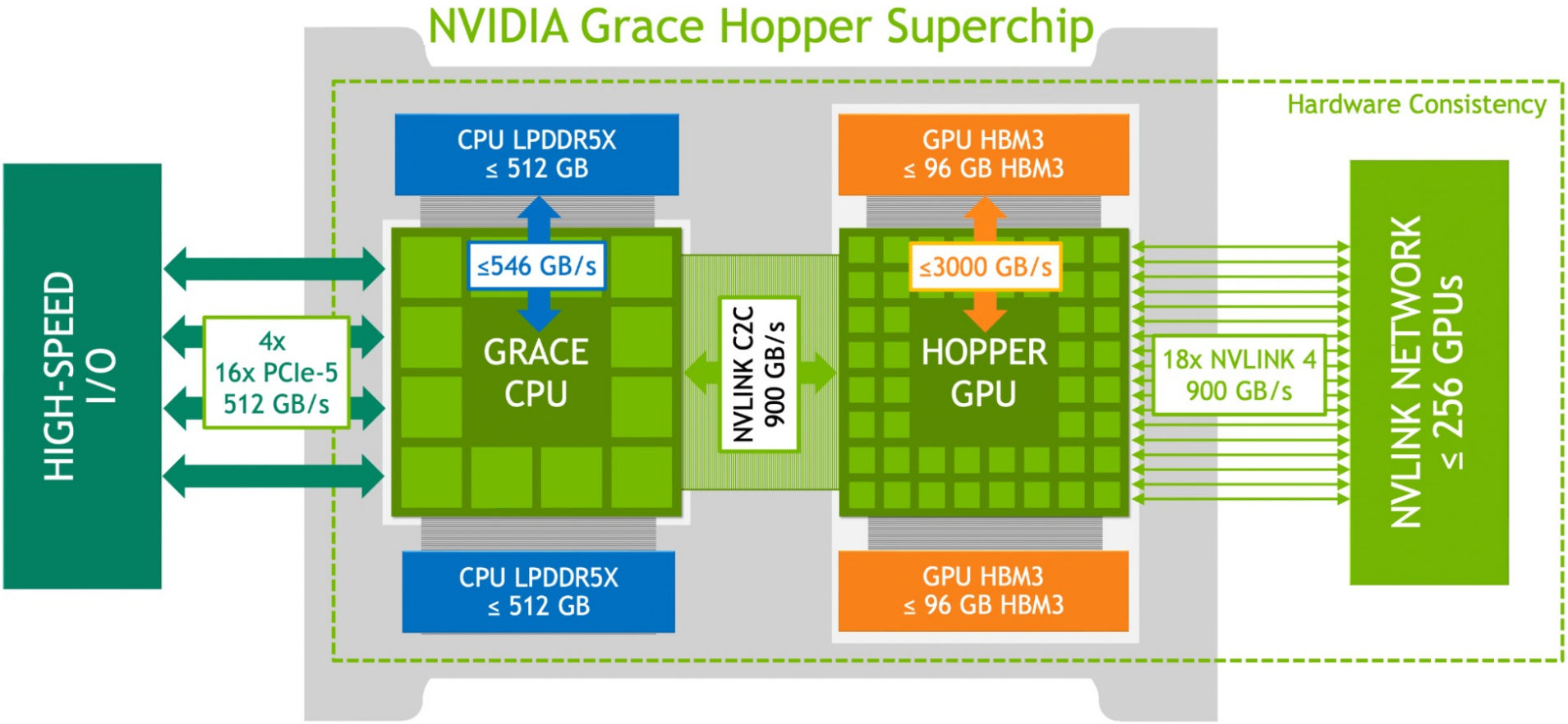
Heterogenous Computing



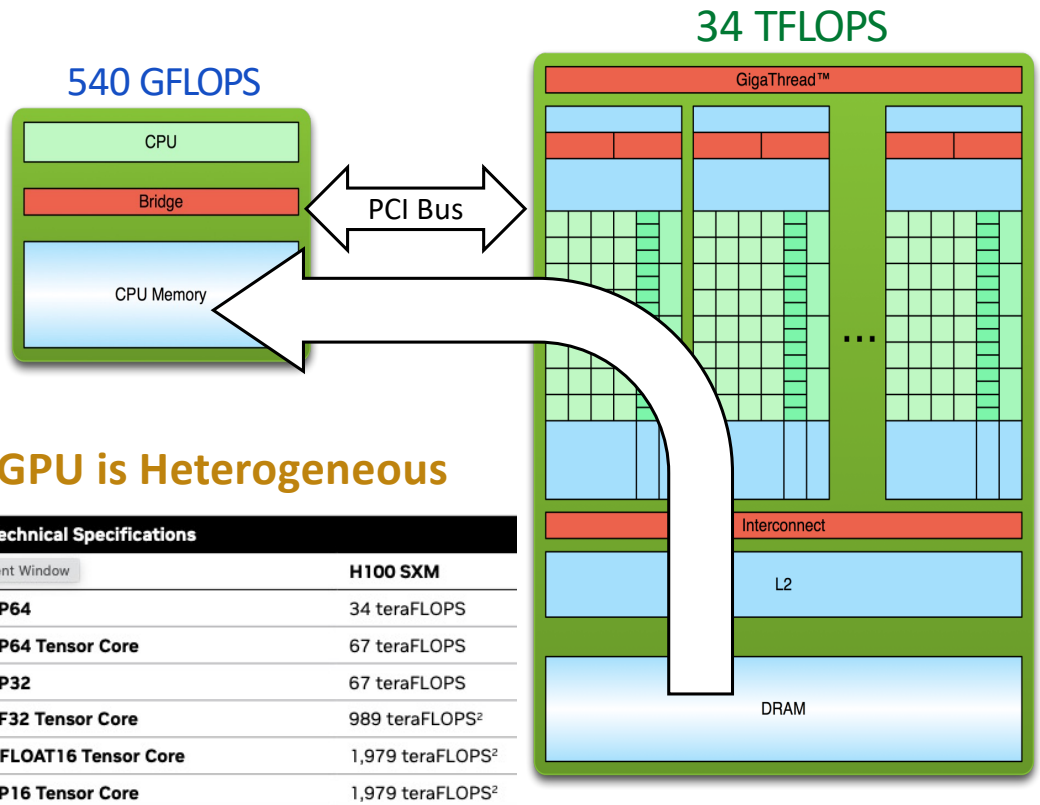


- GPU computing is not meant to replace CPU computing
- CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks
- Modern high-end HPC systems are *heterogenous*: They combine CPUs and GPUs, mapping tasks to the most suitable PU
- A typical heterogeneous compute node consists of two multicore CPU sockets and two or more many-core GPUs
- GPUs operate in conjunction with a CPU-based host typically through a PCI-Express bus

Heterogenous Computing



Heterogenous Computing

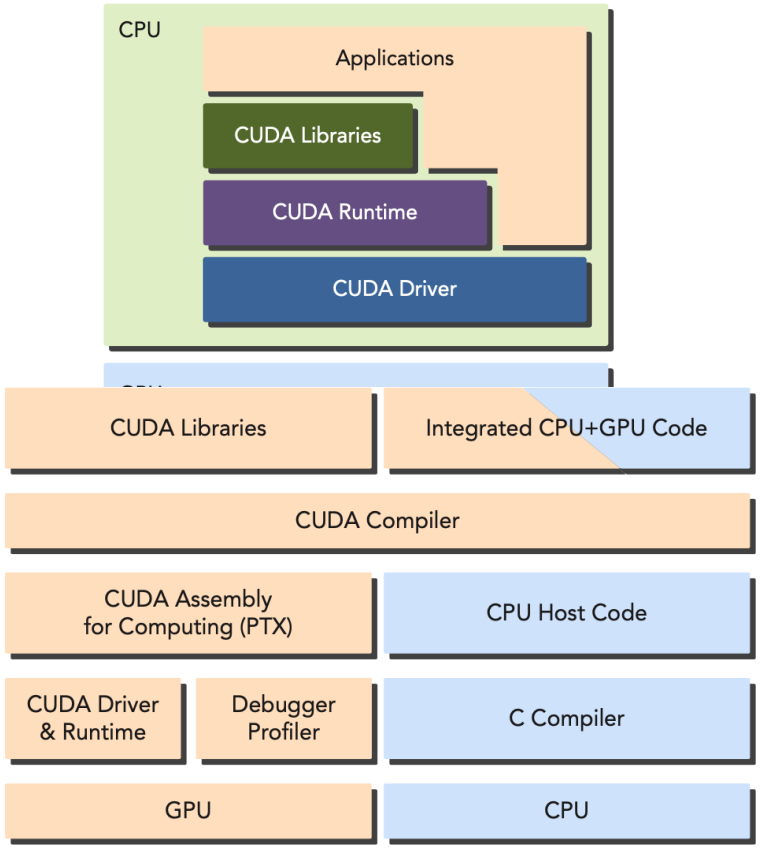


GPU is Heterogeneous

Technical Specifications	
Content Window	**H100 SXM**
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]
FP16 Tensor Core	1,979 teraFLOPS[2]
FP8 Tensor Core	3,958 teraFLOPS[2]
INT8 Tensor Core	3,958 TOPS[2]
GPU memory	80GB
GPU memory bandwidth	3.35TB/s

- In a heterogeneous, the CPU is called the **host** and the GPU is called the **device**
- A heterogeneous application consists of two parts: Host code (runs on CPU) and device code (runs on GPU)
- Applications are initialized by the CPU: the CPU code is responsible for managing the environment, code, and data for the device before loading compute-intensive tasks onto the device.
- **Host and device have distinct and separate virtual memory address spaces**!
- Host ↔ device communication is slow and becomes easily a performance bottleneck.

Compute Unified Device Architecture (CUDA)



- CUDA C is an is an extension of standard ANSI providing APIs and a programming model for NVIDIA GPUs
- A CUDA program consists of a mixture host and device code
- NVIDIA's CUDA `nvcc` compiler separates the device code from the host code during the compilation process
- The device code is written using CUDA C extended with keywords for labeling data-parallel functions, called **kernels**

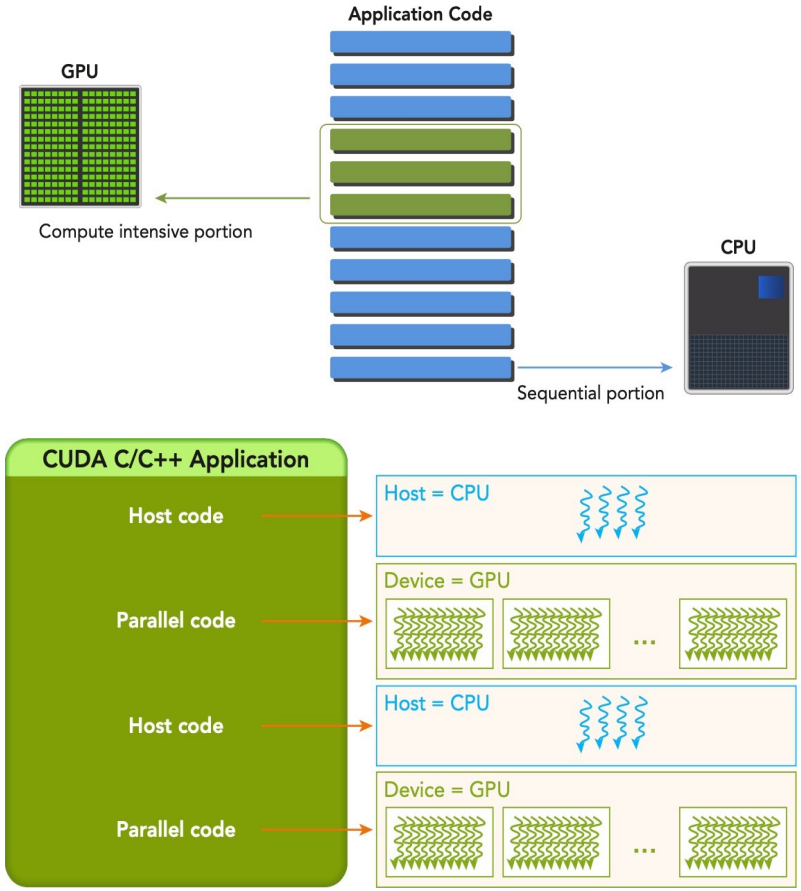
Hello World from a GPU

```
#include <stdio.h>
__global__ void helloFromGPU (void)
{
    printf("Hello World from GPU!\n");
}

int main(void) {
  // hello from cpu
    printf("Hello World from CPU!\n");
    helloFromGPU <<<1, 10>>>();
    cudaDeviceReset();
    return 0;
}
```

```
$ nvcc -arch=sm_70 hello.cu -o hello
$ ./hello
Hello World from CPU!
Hello World from GPU!
Hello World from GPU!
...
Hello World from GPU!
```

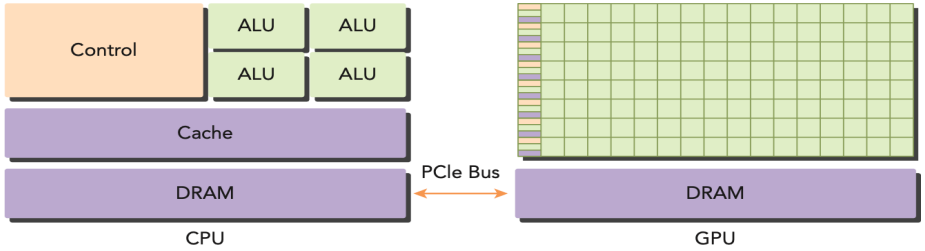
- The qualifier `__global__` tells the compiler the function is a device kernel and will be called from the CPU and executed on the GPU
- The kernel is launched with the triple angle brackets notation (`helloFromGPU <<<1, 10>>>()`)
- The parameters within the triple angle brackets specify how many threads will execute the kernel (10 GPU threads).
- The function `cudaDeviceReset()` cleans up all resources associated with the current device
- The flag `-arch=sm_70` tells the `nvcc` compiler to produce a binary for the Volta V100 architecture

CUDA Programming Structure

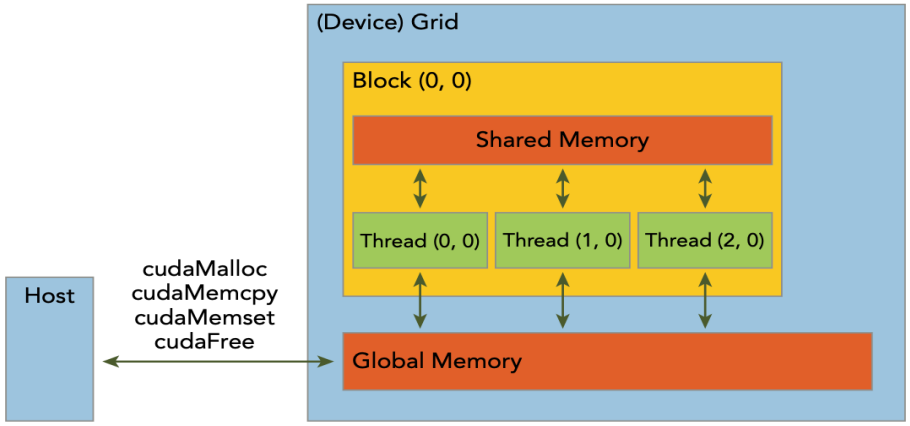


- A typical processing flow of a CUDA program follows this pattern:
  - Copy data from CPU memory to GPU memory
  - Invoke kernels to operate on the data stored in GPU memory
  - Copy data back from GPU memory to CPU memory
- When a kernel has been launched, control is returned immediately to the host.
- The host can operate independently of the device for most operations. CUDA is an asynchronous model.

CUDA Memory Management



STANDARD C FUNCTIONS	CUDA C FUNCTIONS
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`



- CUDA provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory
- GPU memory allocation → synchronous

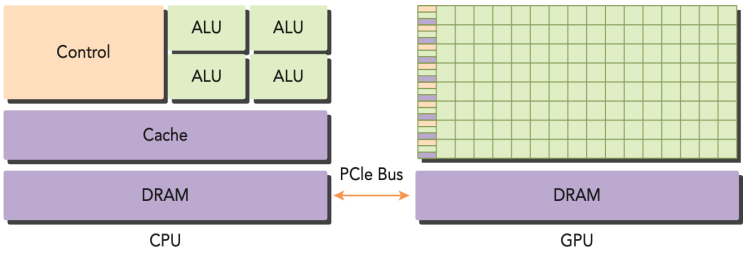
```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

- Transfer data between the host and device → synchronous

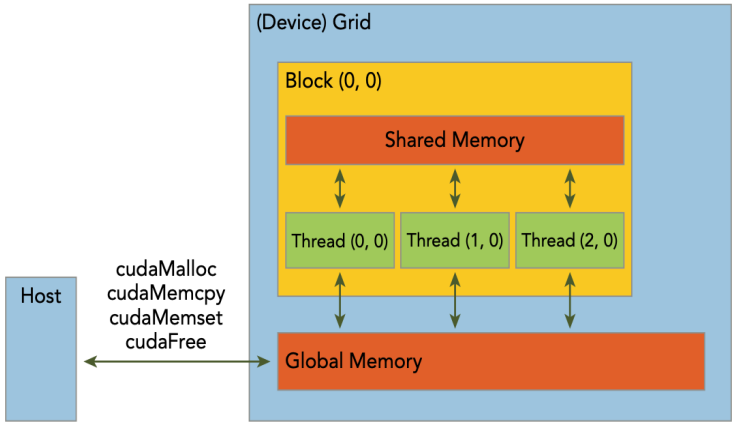
```
cudaError_t cudaMemcpy ( void *  dst,
      const void * src,  size_t  count,
      cudaMemcpyKind  kind )
```

- Kinds of transfer: `cudaMemcpyKind = { cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyDeviceToDevice }`
- `cudaMemset` and `cudaFree` are also synchronous

CUDA Memory Management



STANDARD C FUNCTIONS	CUDA C FUNCTIONS
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

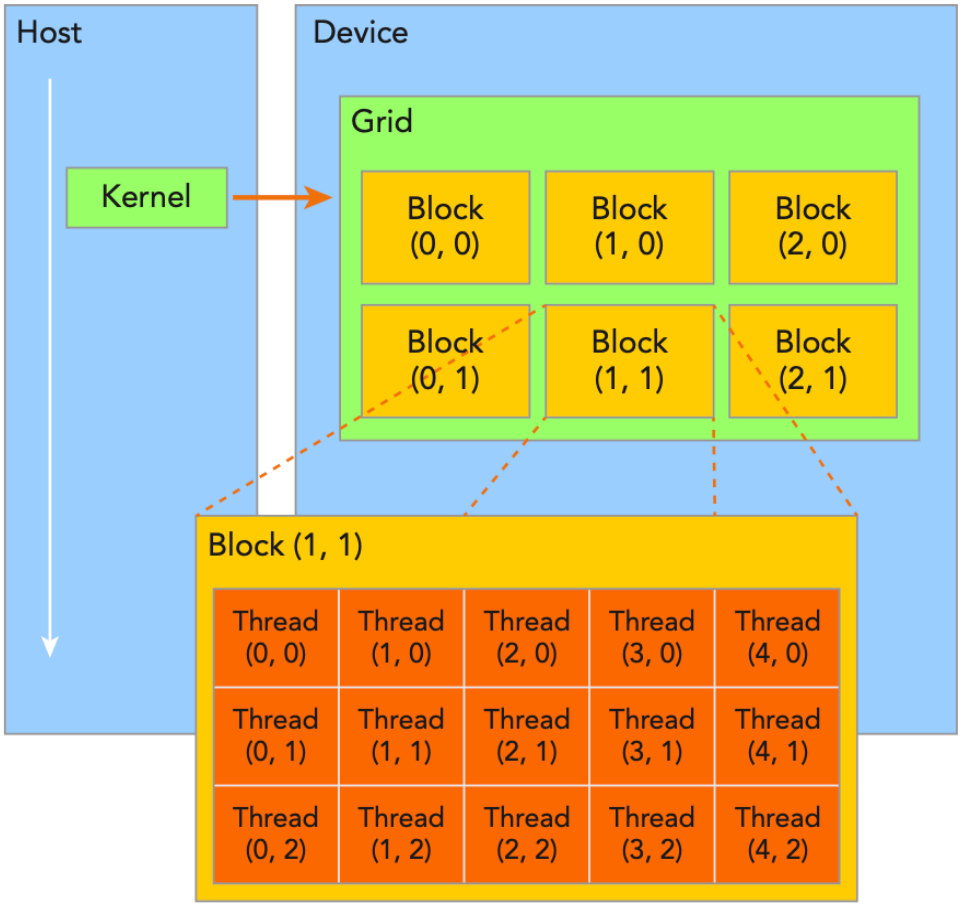


- CUDA provides functions to allocate device memory, release device memory, and transfer data between the host memory and device memory
- GPU memory allocation → synchronous

```
cudaError_t cudaMalloc ( void** devPtr, size_t size )
```

- **WARNING**: device pointers (e.g `devPtr`) may not be dereferenced in the host code.

CUDA Thread Organization



- Two-level thread hierarchy decomposed into blocks of threads and grids of blocks
- All threads spawned by a single kernel form a thread *grid*
- Threads in a grid are grouped in thread blocks
- Threads in the same block can cooperate using *block-local sychronization* and *shared memory*
- Threads from different blocks cannot synchronize!
- Each block has a unique ID, `bblockIdx`, within the grid
- Each thread has a unique ID, `threadIdx`, within its block (local)

Defining Grids and Blocks

```
int nElem = 6;
// define grid and block structure dim3
block (3);
dim3 grid ((nElem+block.x-1)/block.x);
// check grid and block dimension from host side
printf("grid.x %d grid.y %d grid.z %d\n",
        grid.x, grid.y, grid.z);
printf("block.x %d block.y %d block.z %d\n",
        block.x, block.y, block.z);
// check grid and block dimension from device
    side
 checkIndex <<<grid, block>>> ();

 __global__void checkIndex(void) {

 printf("threadIdx:(%d, %d, %d) blockIdx:(%d, %d,
    %d) blockDim:(%d, %d, %d) gridDim:(%d, %d,
    %d)\n",
  threadIdx.x, threadIdx.y, threadIdx.z,
  blockIdx.x, blockIdx.y, blockIdx.z, block
  Dim.x, blockDim.y, blockDim.z, gridDim.x,
  gridDim.y, gridDim.z);
}
```

```
grid.x 2 grid.y 1 grid.z 1
block.x 3 block.y 1 block.z 1
threadIdx:(0, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0)  blockIdx:(1, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(0, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(1, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
threadIdx:(2, 0, 0)  blockIdx:(0, 0, 0) blockDim:(3, 1, 1) gridDim:(2, 1, 1)
```

- CUDA organizes grids and blocks in three dimensions
- `uint3 blockIdx = {blockIdx.x, blockIdx.y, blockIdx.z}`
- `uint3 threadIdx = {threadIdx.x, threadIdx.y, threadIdx.z}`
- When defined on the host grids and blocks use the `dim3` type (and not `uint3`) with 3 unsigned integer fields
- Note that the grid size is rounded up to the multiple of the block size
- For a given kernel, the grid and block dimensions are decided based on performance characteristics and limitations of GPU resources

CUDA Kernel Semantics

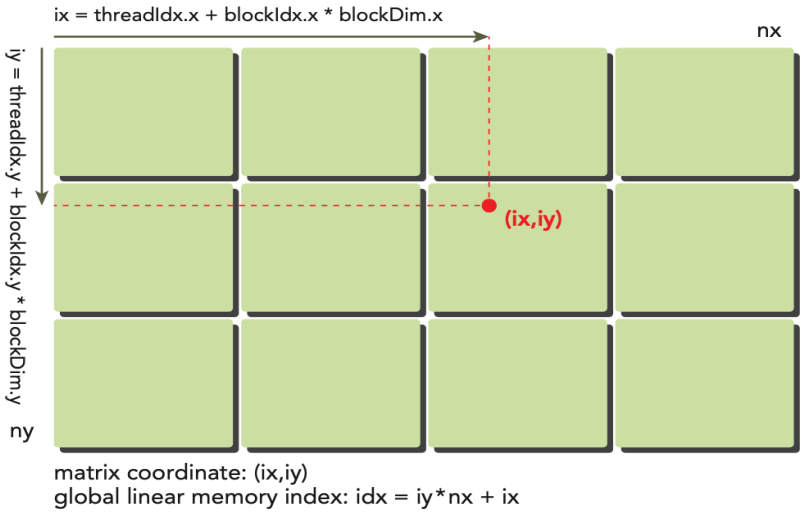
- The definition of a CUDA kernel requires special function qualifiers
  - `__global__` → Executed on device, callable from host and device, must have `void` return type
  - `__device__` → Executed on device, callable from device only
  - `__host__` → Executed on host, callable from host only
- GPU kernels use implicit parallelism!
- For example, from the host code

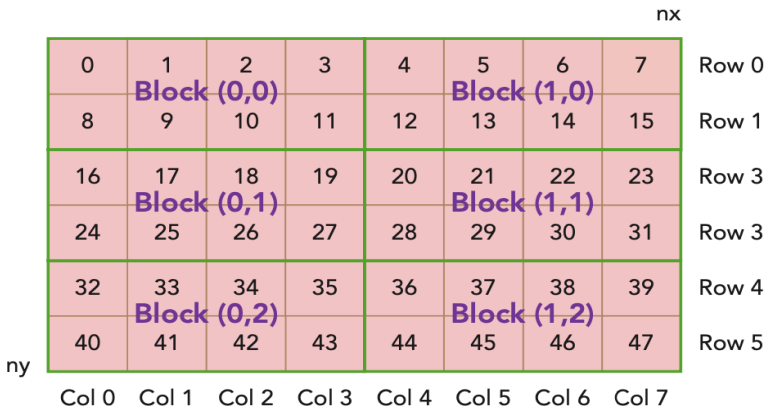
```
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
  for (int i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
  }
}
```

- You can obtain a GPU parallel kernel by peeling off the `for` loop and assigning work to different threads

```
_global__ void sumArraysOnGPU(float *A, float *B, float *C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

Organizing Threads: Matrix Addition





- We want to perform the matrix sum $C = A + B$ in parallel on the GPU.
- The matrices have dimensions *nx* and *ny*
- Each thread performs the addition

  $C(ix, iy) = A(ix, iy) + B(ix, iy)$

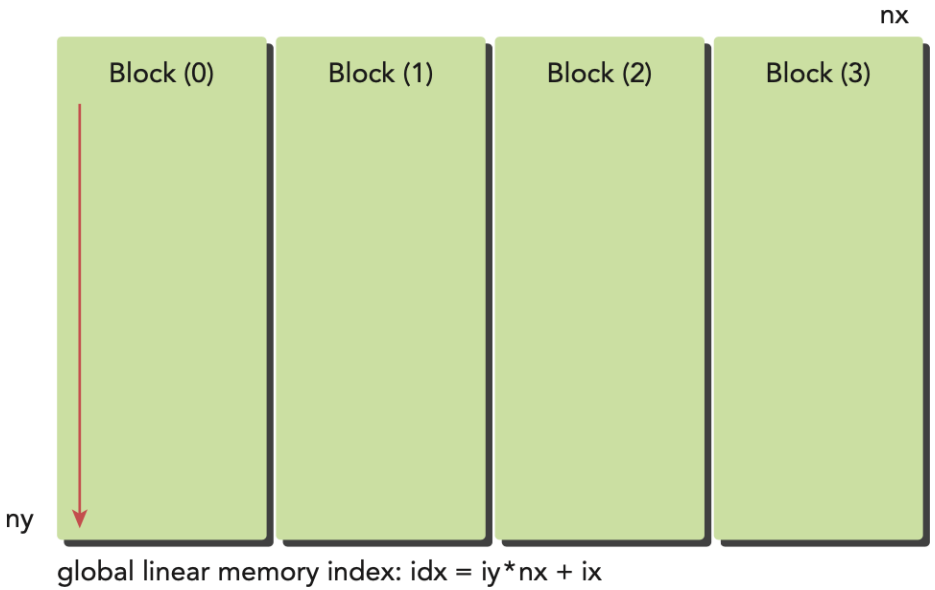
  for a distinct element of A, B and C with row and column indices (*ix, iy*)
- We can map a single thread to each matrix element in the A, B or C arrays at position `idx` using a 2D grid of thread blocks where
  - `ix = threadIdx.x + blockIdx.x * blockDim.x`
  - `ix = threadIdx.y + blockIdx.y *`
  - `blockDim.y`
  - `idx = iy * nx + ix`

Matrix Addition with 2D Grid and 2D Blocks

- Matrix dimensions nx = ny = 16,384
- Kernel execution configuration set to use a 2D grid and 2D block between lines 9-12
- Running on an NVIDIA Kepler K80
  - `sumMatrixOnGPU2D <<<(512,512), (32,32)>>> elapsed 0.060323 sec`
  - `sumMatrixOnGPU2D <<<(512,1024), (32,16)>>> elapsed 0.038041 sec`
  - `sumMatrixOnGPU2D <<< (1024,1024), (16,16) >>> elapsed 0.045535 sec`

```
  // malloc device global memory
float *d_MatA, *d_MatB, *d_MatC;
cudaMalloc((void **)&d_MatA, nBytes);
cudaMalloc((void **)&d_MatB, nBytes);
cudaMalloc((void **)&d_MatC, nBytes)
// transfer data from host to device
cudaMemcpy(d_MatA, h_A, nBytes,
    cudaMemcpyHostToDevice);
cudaMemcpy(d_MatB, h_B, nBytes,
    cudaMemcpyHostToDevice);
// invoke kernel at host side
int dimx = 32; int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx+block.x-1)/block.x, (ny+block.y-1)/
    block.y);
iStart = cpuSecond();
sumMatrixOnGPU2D <<< grid, block >>>(d_MatA,
    d_MatB, d_MatC, nx, ny);
    cudaDeviceSynchronize();
iElaps = cpuSecond() - iStart;

__global__ void sumMatrixOnGPU2D(float *MatA,
    float *MatB, float *MatC, int nx, int ny) {
unsigned int ix = threadIdx.x + blockIdx.x *
    blockDim.x;
unsigned int iy = threadIdx.y + blockIdx.y *
    blockDim.y;
unsigned int idx = iy*nx + ix;
if (ix < nx && iy < ny)
  MatC[idx] = MatA[idx] + MatB[idx];
}
```

Matrix Addition with 1D Grid and 1D Blocks

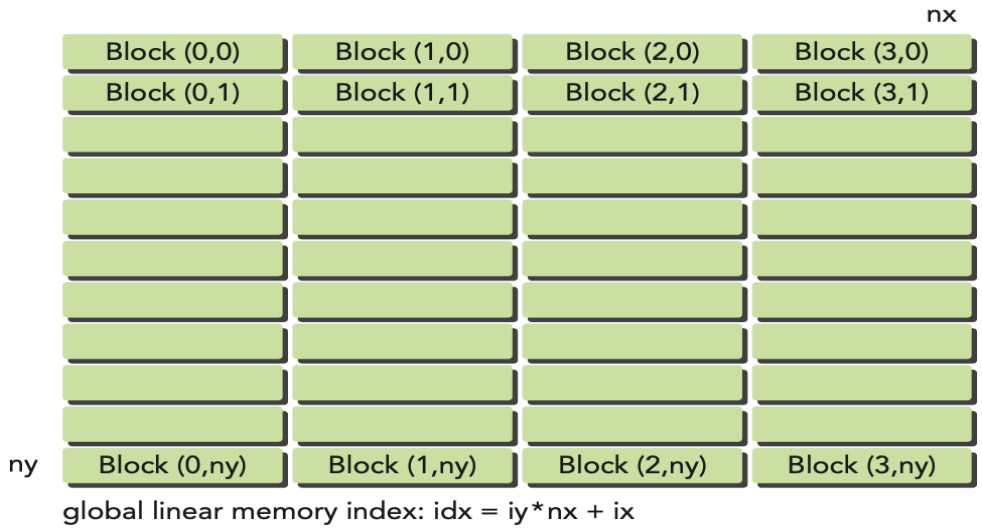



```
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB
    , float *MatC, int nx, int ny) {
unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
if (ix < nx ) {
  for (int iy=0; iy<ny; iy++) {
    int idx = iy*nx + ix;
    MatC[idx] = MatA[idx] + MatB[idx];
  }
}
}
```

- Matrix dimensions *nx* = *ny* = 16,384
- Now we use a 1D grid with 1D blocks
- Each thread in the new kernel handles ny elements
- Running on an NVIDIA Kepler K80

- `sumMatrixOnGPU1D <<<(512,1), (32,1)>>>`
  `elapsed 0.061352 sec`
- `sumMatrixOnGPU1D <<<(128,1),(128,1)>>>`
  `elapsed 0.044701 sec`

Matrix Addition with 2D Grid and 1D Blocks




```
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *
    MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x *blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy*nx + ix;
if (ix < nx && iy < ny)
  MatC[idx] = MatA[idx] + MatB[idx];
}
```

- Now we use a 2D grid with 1D blocks
- Each thread takes care of only one data element and the second dimension of grid equals ny
- Running on an NVIDIA Kepler K80
  - `sumMatrixOnGPUMix <<<(512,16384), (32,1)>>> elapsed 0.073727 s`
  - `sumMatrixOnGPUMix <<<(64,16384), (256,1)>>> elapsed 0.030765 s` (best performance so far)
- Changing execution configurations affects performance
- A naive kernel implementation does not generally yield the best performance
- For a given kernel, trying different grid and block dimensions may yield better performance