# GPGPUs and CUDA

Abhinav Bhatele, Alan Sussman

UNIVERSITY OF
MARYLAND

Many slides borrowed from Daniel Nichols' slides

# Announcements

- Assignment 2 on MPI (heat equation) due next Friday, March 7, 7PM

  - Questions?

-

DEPARTMENT OF
COMPUTER SCIENCE

# GPGPUs

- Originally developed to handle computation related to graphics processing

- Also found to be useful for scientific computing

- Hence the name: General Purpose Graphics Processing Unit

DEPARTMENT OF
COMPUTER SCIENCE

# Accelerators

- IBM's Cell processors

  - Used in Sony's Playstation 3 (2006)

- GPUs: NVIDIA, AMD, Intel

  - First programmable GPU: NVIDIA GeForce 256 (1999)

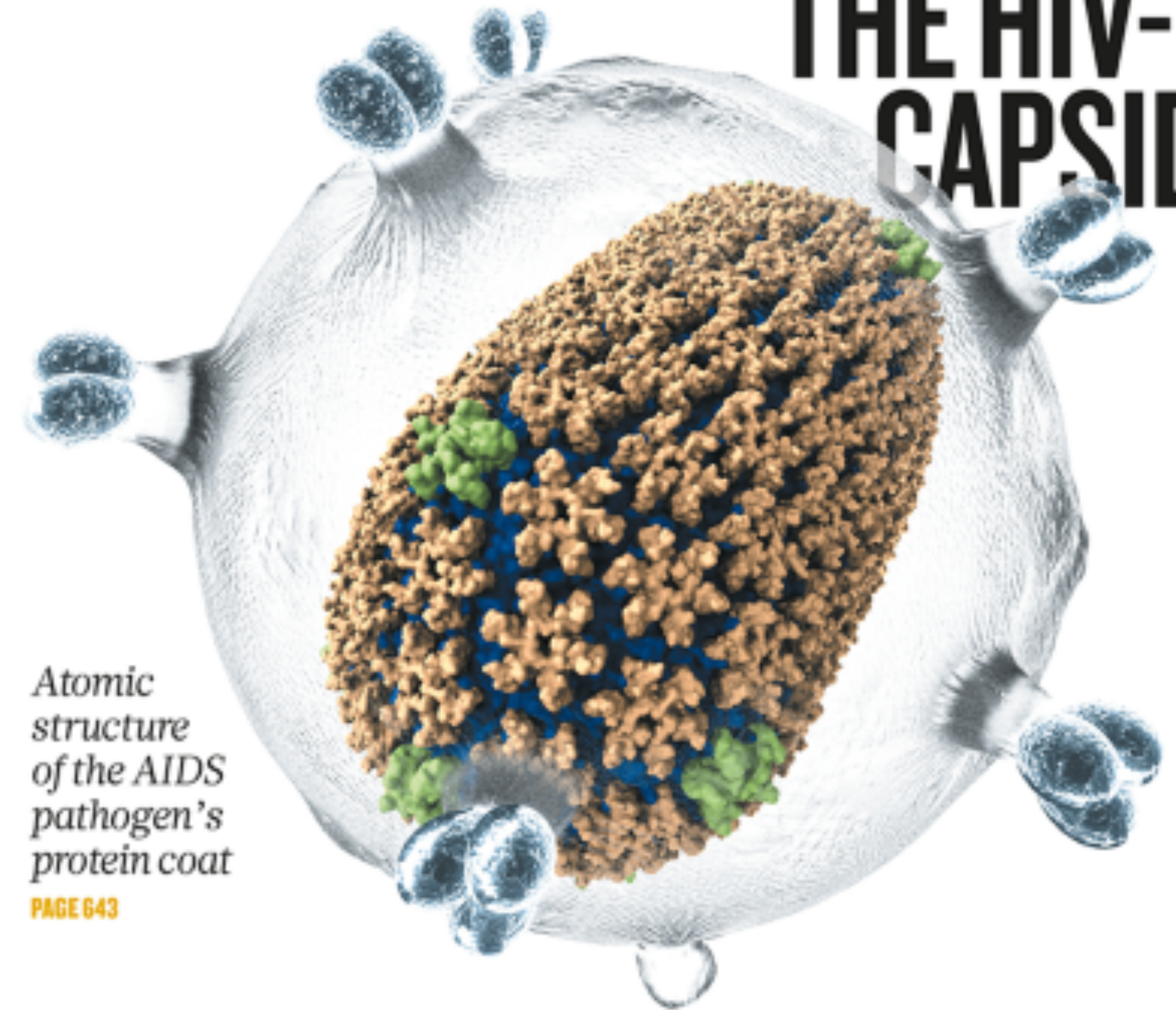  - Around 1999-2001, early GPGPU results

- FPGAs

https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf

# Used for mainstream HPC



- 2013: NAMD, used for molecular dynamics simulations on a supercomputer with 3000 NVIDIA Tesla GPUs
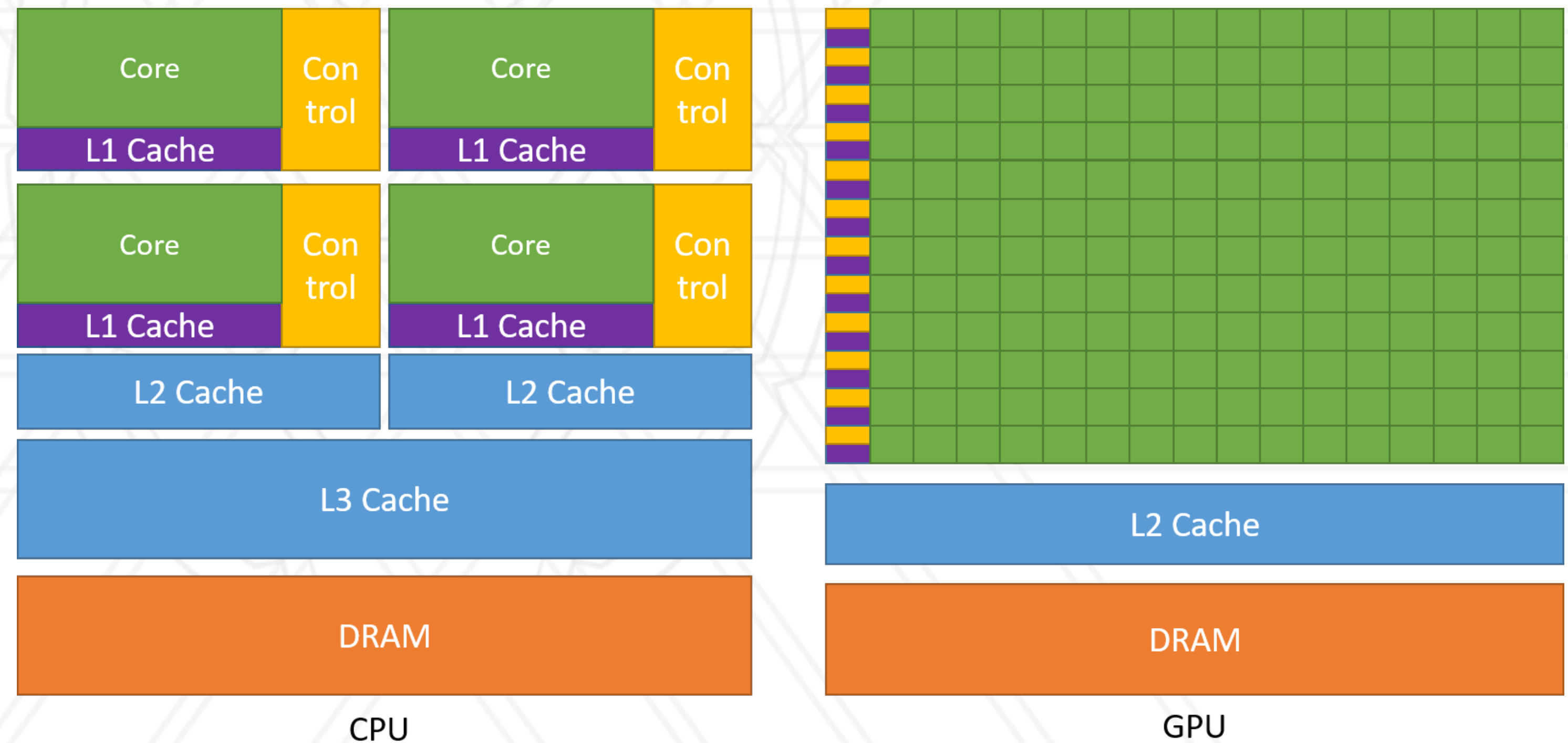
# GPGPU Hardware

- Higher instruction throughput

- Hide memory access latencies with computation

# Comparing GPUs to CPUs

- Intel i9 11900K

  - 8 cores

  - 3.3 GHz

- AMD Epic 7763

  - 64 cores

  - 2.45 GHz

- NVIDIA GeForce RTX 3090

  - 10,496 cores

  - 1.4 GHz

- NVIDIA A100

  - 17,712 cores

  - 0.76 GHz

DEPARTMENT OF
COMPUTER SCIENCE

# Volta GV100 SM

- CUDA Core

  - Single serial execution unit

- Each Volta Streaming Multiprocessor (SM) has:

  - 64 FP32 cores

  - 64 INT32 cores

  - 32 FP64 cores

  - 8 Tensor cores

- CUDA capable device or GPU

  - Collection of SMs

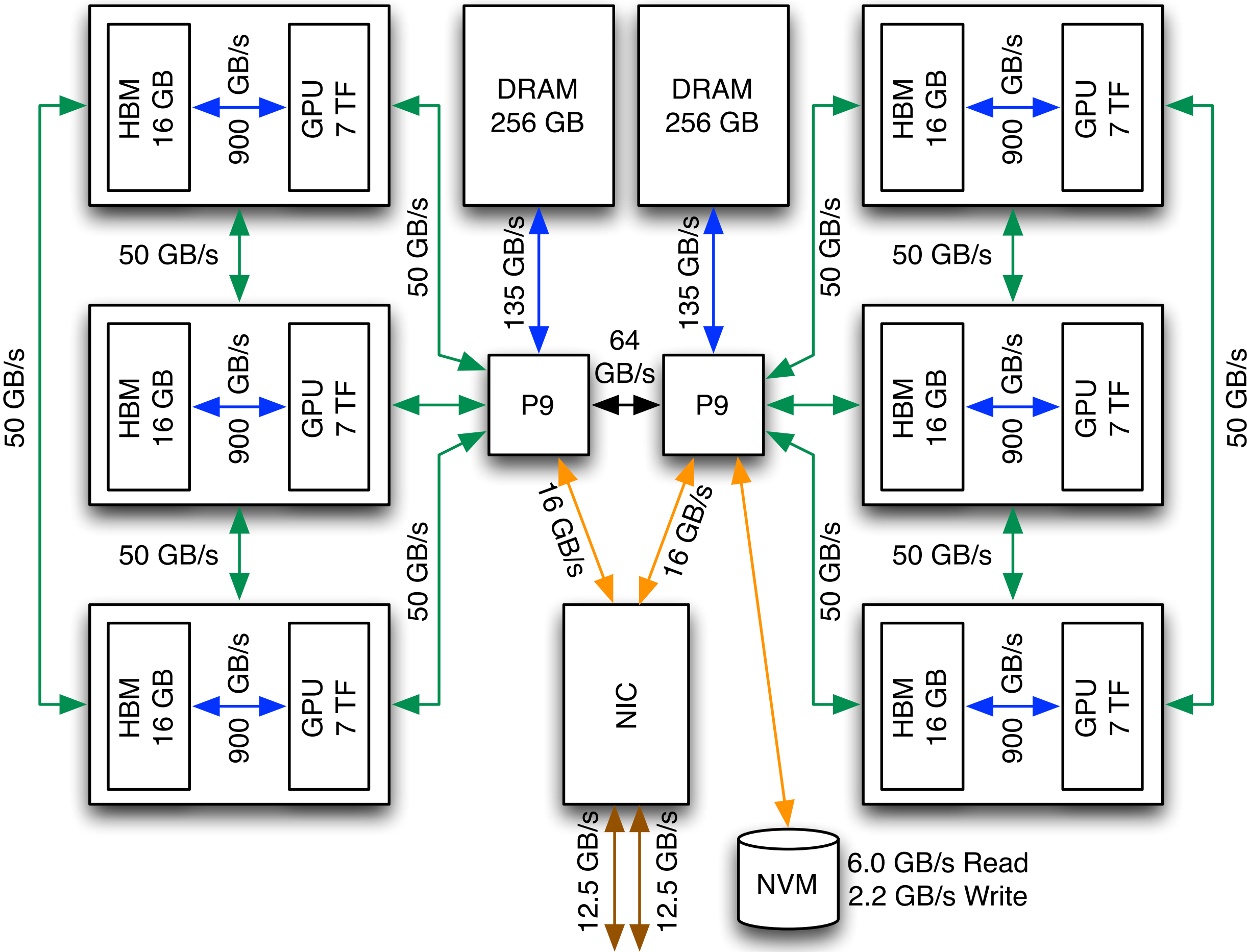https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

DEPARTMENT OF
COMPUTER SCIENCE

# Vo

# GPU-based nodes

- Figure on the right shows a single node of Summit @ ORNL



| | | |
|---|---|---|
| TF | 42 TF (6x7 TF) | |
| HBM | 96 GB (6x16 GB) | |
| DRAM | 512 GB (2x16x16 GB) | |
| NET | 25 GB/s (2x12.5 GB/s) | |
| MMsg/s | 83 | |

HBM/DRAM Bus (aggregate B/W)
NVLINK
X-Bus (SMP)
PCIe Gen4
EDR IB

HBM & DRAM speeds are aggregate (Read+Write).
All other speeds (X-Bus, NVLink, PCIe, IB) are bi-directional.

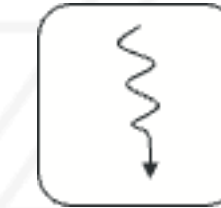# CUDA: A programming model for NVIDIA GPUs

- Allows developers to use C++ as a high-level programming language

- Built around threads, blocks and grids

- Terminology:

  - Host: CPU

  - Device: GPU

  - CUDA kernel: a function that gets executed on the GPU

```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```
CUDA Kernel

CUDA threads

# CUDA software abstraction

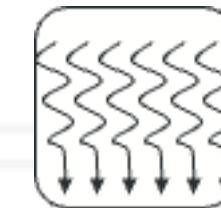- ## Thread

  - Serial unit of execution

- ## Block

  - Collection of threads

  - Number of threads in block <= 1024

- ## Grid

  - Collection of blocks

DEPARTMENT OF
COMPUTER SCIENCE

# Software to hardware mapping



https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/

# Three steps to writing a CUDA kernel

- Copy input data from host to device memory

- Load the GPU program (kernel) and execute

- Copy the results back to host memory

DEPARTMENT OF
COMPUTER SCIENCE

# Copying data to the GPU

```
double *d_Matrix, *h_Matrix;
h_Matrix = new double[N];

cudaMalloc(&d_Matrix, sizeof(double)*N);

// ... initialize h_Matrix ...
cudaMemcpy(d_Matrix, h_Matrix, sizeof(double)*N, cudaMemcpyHostToDevice);

// ... some computation on GPU …

cudaMemcpy(h_Matrix, d_Matrix, sizeof(double)*N, cudaMemcpyDeviceToHost);

cudaFree(d_Matrix);
```

cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice
cudaMemcpyHostToHost
cudaMemcpyDefault

DEPARTMENT OF
COMPUTER SCIENCE

# CUDA syntax

```
__global__ void saxpy(float *x, float *y, float alpha) {
    int i = threadIdx.x;
    y[i] = alpha*x[i] + y[i];
}

int main() {
    ...
    saxpy<<<1, N>>>(x, y, alpha);
    ...
}
```

What happens when:
array size (N) > 1024?

`<<<#blocks, threads_per_block>>>`
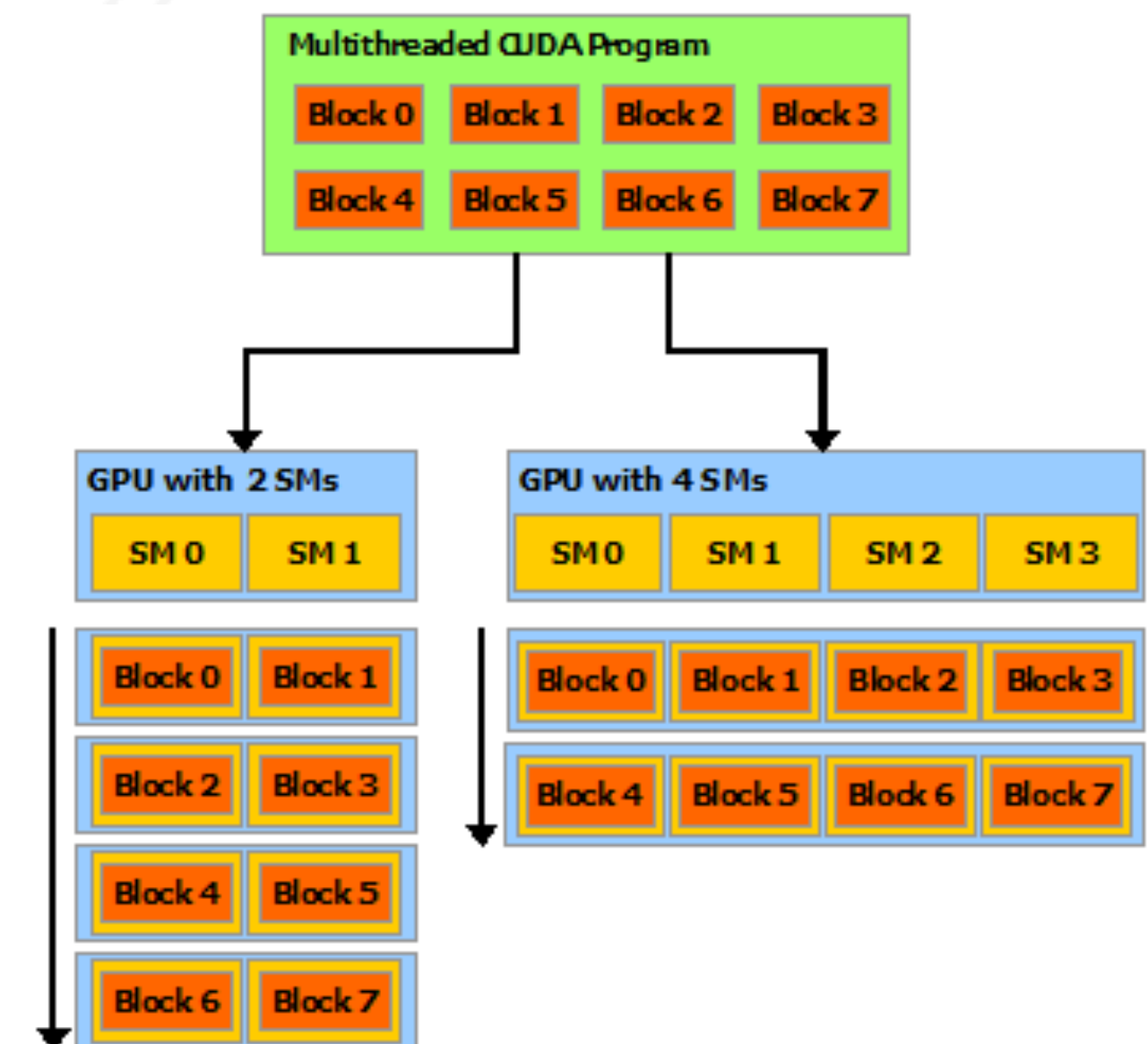
DEPARTMENT OF
COMPUTER SCIENCE

# Compiling and Running CUDA code

```
nvcc -o saxpy --generate-code arch=compute_80,code=sm_80 saxpy.cu

./saxpy
```

DEPARTMENT OF
COMPUTER SCIENCE

# Multiple blocks

```
__global__ void saxpy(float *x, float *y, float alpha, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        y[i] = alpha*x[i] + y[i];
}



int main() {
    ...
    int threadsPerBlock = 512;
    int numBlocks = N/threadsPerBlock
                    + (N % threadsPerBlock != 0);

    saxpy<<<numBlocks, threadsPerBlock>>>(x, y, alpha, N);
    ...
}
```
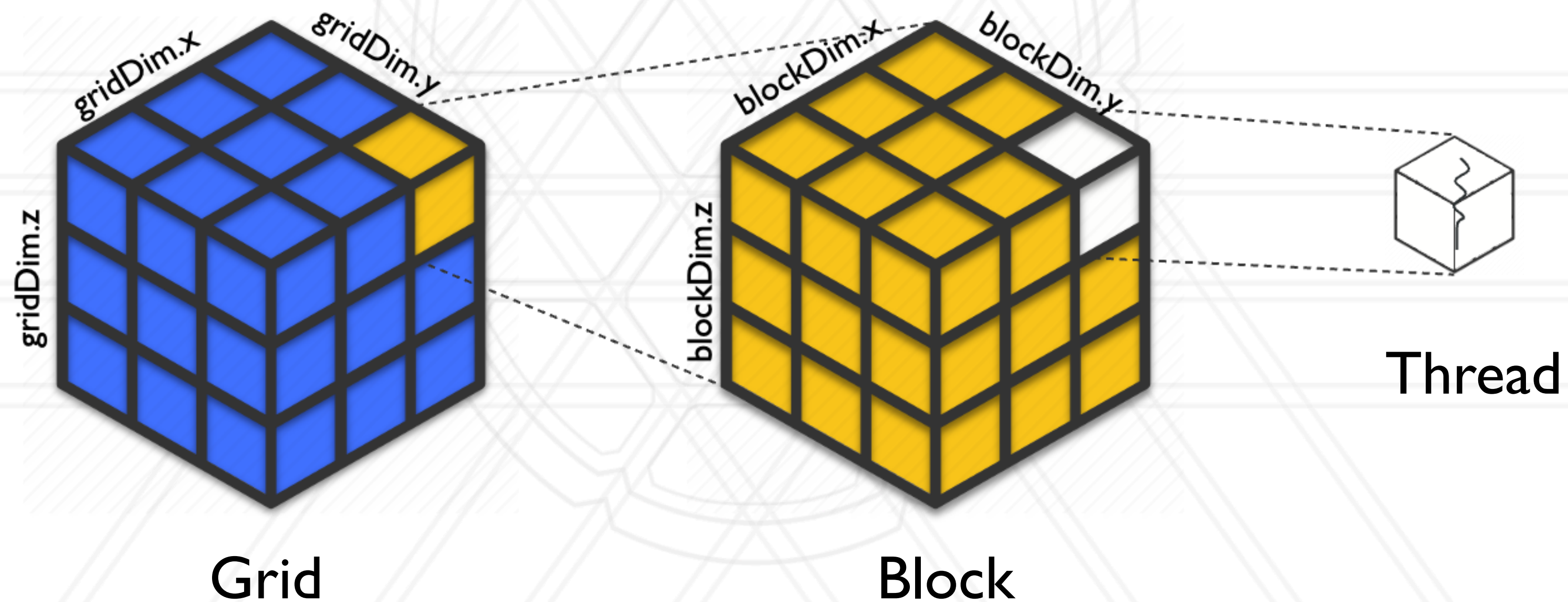
# Striding

```
__global__ void saxpy(float *x, float *y, float alpha, int N) {
    int i0 = blockDim.x * blockIdx.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = i0; i < N; i += stride)
        y[i] = alpha*x[i] + y[i];
}


int main() {
    ...
    int blockSize = 8; int gridSize = 16;

    saxpy<<<gridSize, blockSize>>>(x, y, alpha, N);
    ...
}
```

# Grid and Block Dimensions

- Blocks in a grid and threads in a block can be arranged in a 3D virtual mesh

- And they can be indexed in 3D



Grid        Block        Thread

# Adding 2D matrices

```
__global__ void matrixAdd(float **X, float **Y, float alpha, int M, int N) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;

    if (i < M && j < N)
        Y[i][j] = alpha * X[i][j] + Y[i][j];
}

int main() {
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(M/threadsPerBlock.x + (M % threadsPerBlock.x != 0),
            N/threadsPerBlock.y + (N % threadsPerBlock.y != 0));

    matrixAdd<<<numBlocks, threadsPerBlock>>>(X, Y, alpha, M, N);
}
```
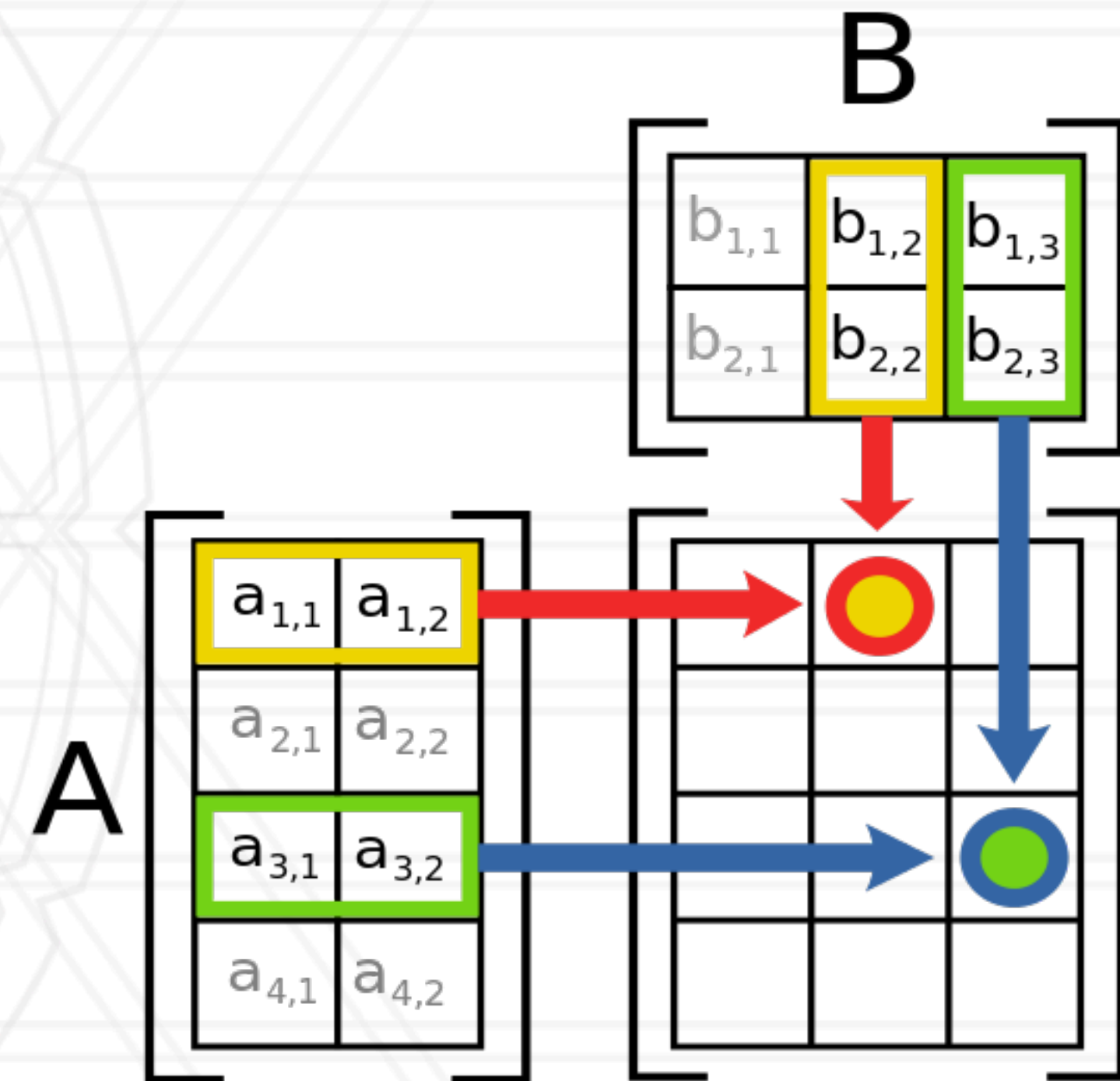
# Matrix Multiply in CUDA

- Solution: use 2D virtual mesh of threads

- Each thread computes one element of the C matrix

- Thread $(i, j)$ computes $c_{ij}$



B

A

https://en.wikipedia.org/wiki/Matrix_multiplication

```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
```

# Matrix Multiply in CUDA

```
int main() {
    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlocks(M/threadsPerBlock.x + (M % threadsPerBlock.x != 0),
            N/threadsPerBlock.y + (N % threadsPerBlock.y != 0));

    matmul<<<numBlocks, threadsPerBlock>>>(C, A, B, M, L, N);
}
```
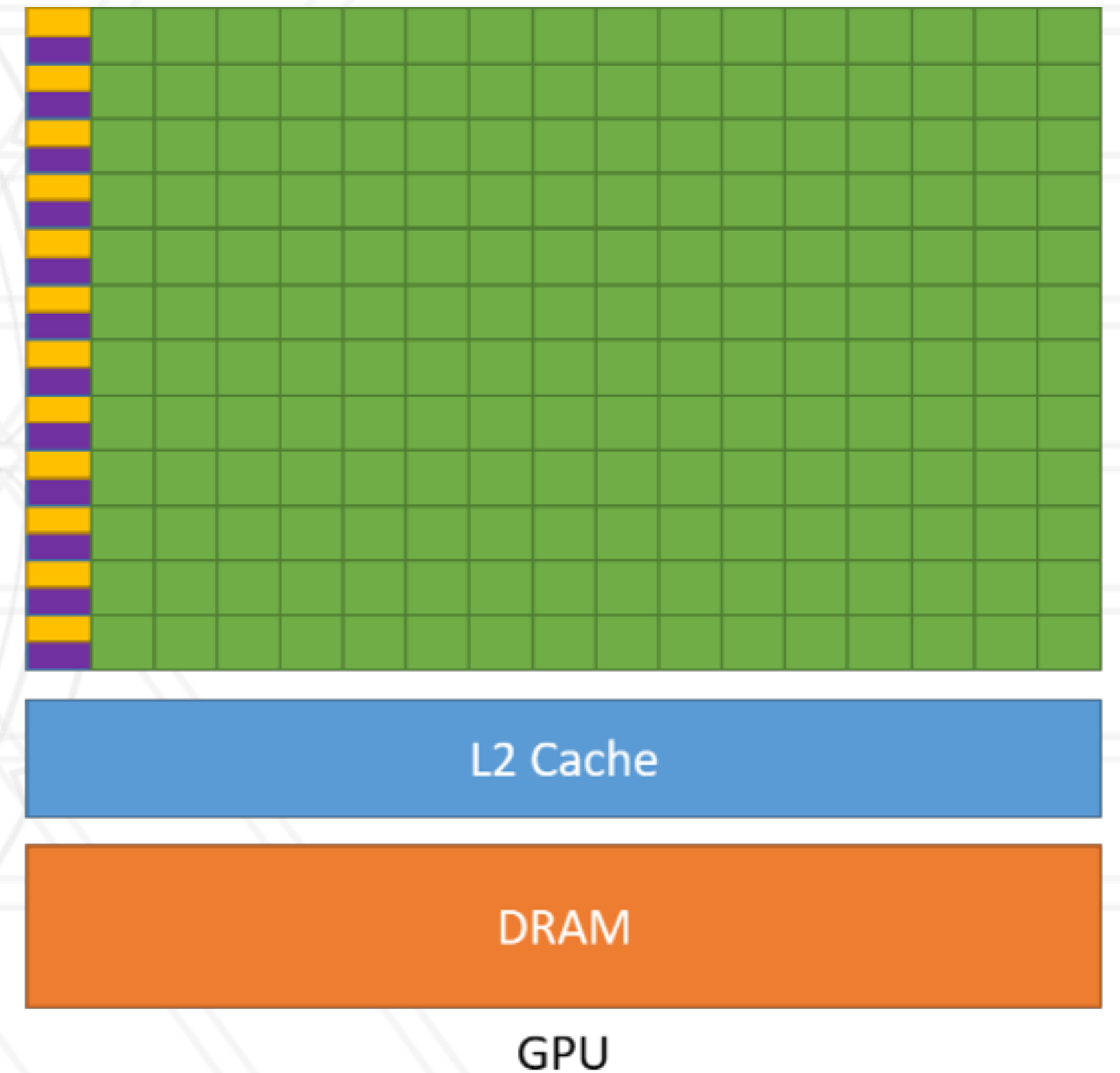
# Matrix Multiply in CUDA

```
__global__ void matmul(double *C, double *A, double *B, size_t M,
size_t L, size_t N) {

    int i = blockDim.x*blockIdx.x + threadIdx.x;
    int j = blockDim.y*blockIdx.y + threadIdx.y;

    if (i < M && j < N) {
        for (int k = 0; k < L; k++) {
            C[i*N+j] += A[i*L+k] * B[k*N+j];
        }
    }
}
```

DEPARTMENT OF
COMPUTER SCIENCE

# Any potential performance issues?

- Poor data re-use

- Every value of A & B is loaded from the global DRAM

- Both A and B are read multiple times



L2 Cache

DRAM

GPU

# Different types of GPU memory

- Global memory: this is like the main memory on a CPU

  - Allocated using cudaMalloc

- Shared memory: very fast memory located in the SMs

  - All threads in a block can access this shared memory

  - Allocated in the kernel using `__shared__`

- Local memory: everything on the stack that can't fit in registers

  - Stored in global memory
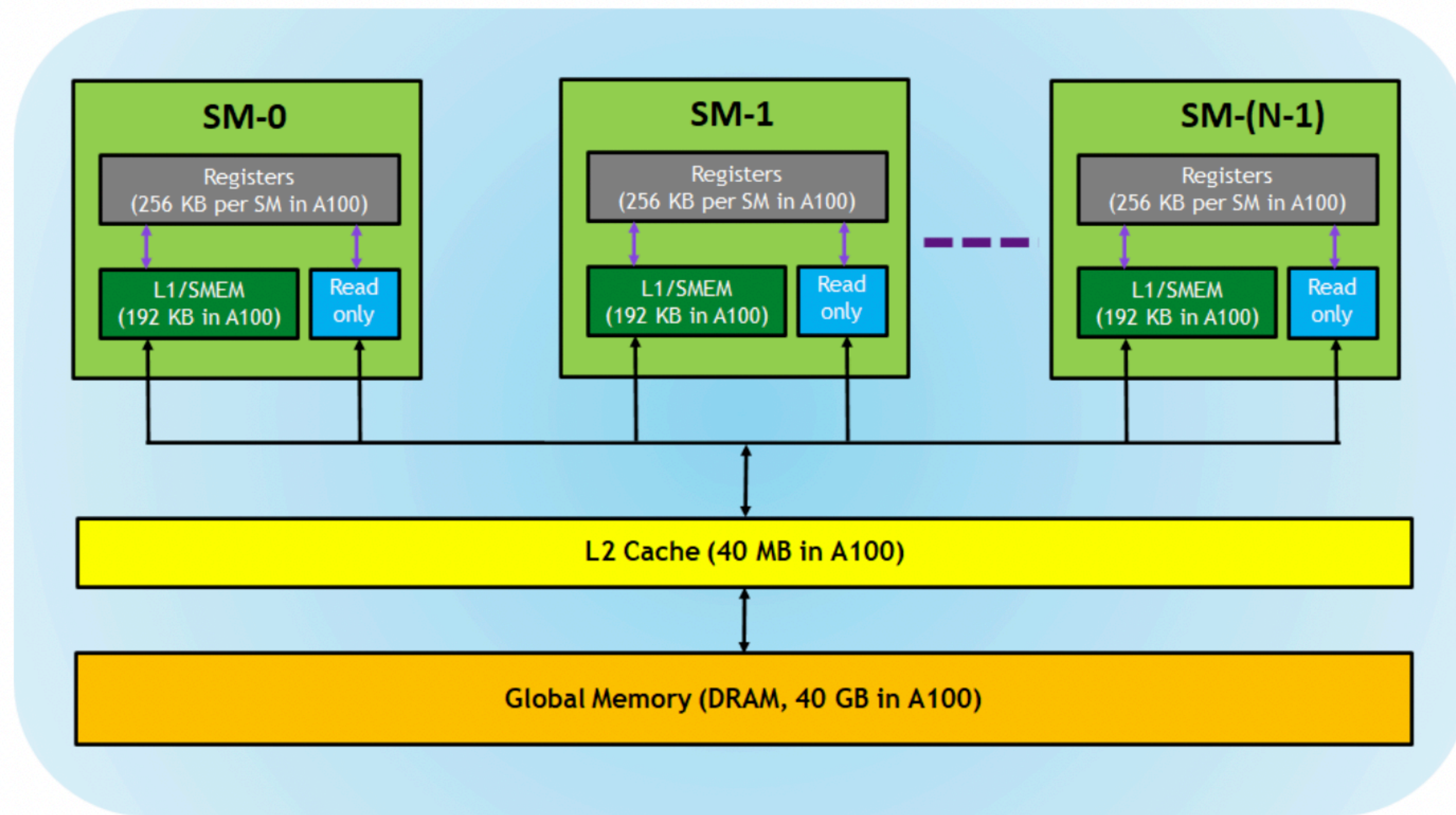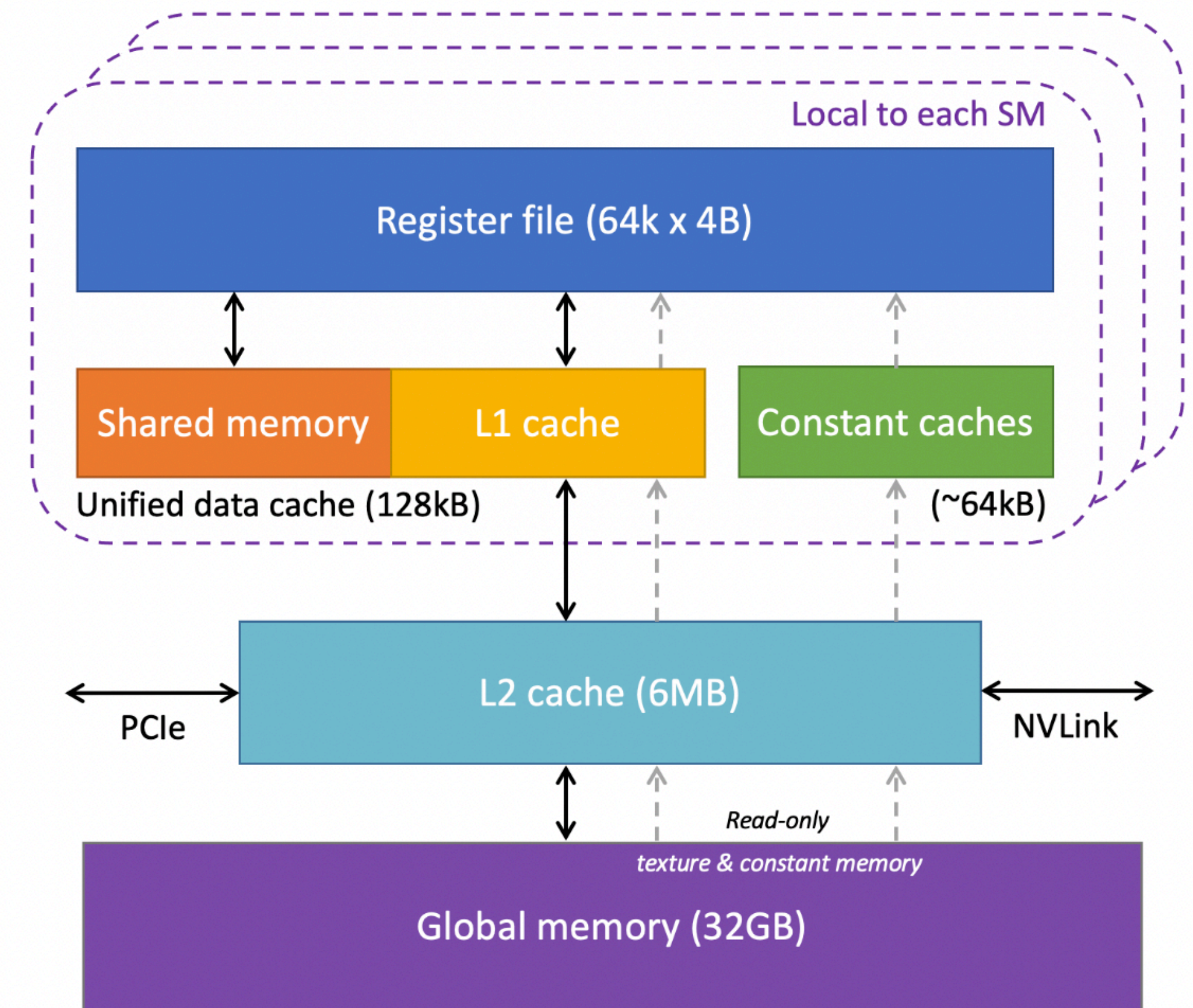
  - Private to each thread

DEPARTMENT OF
COMPUTER SCIENCE

# Different types of GPU memory



Figure 4. Memory hierarchy in GPUs.

https://velog.io/@hansoljang/GPU-Architecture-and-Memory



GPU memory levels and sizes for the NVIDIA Tesla V100.
(Based on diagrams by NVIDIA and Citadel at GTC 2018)

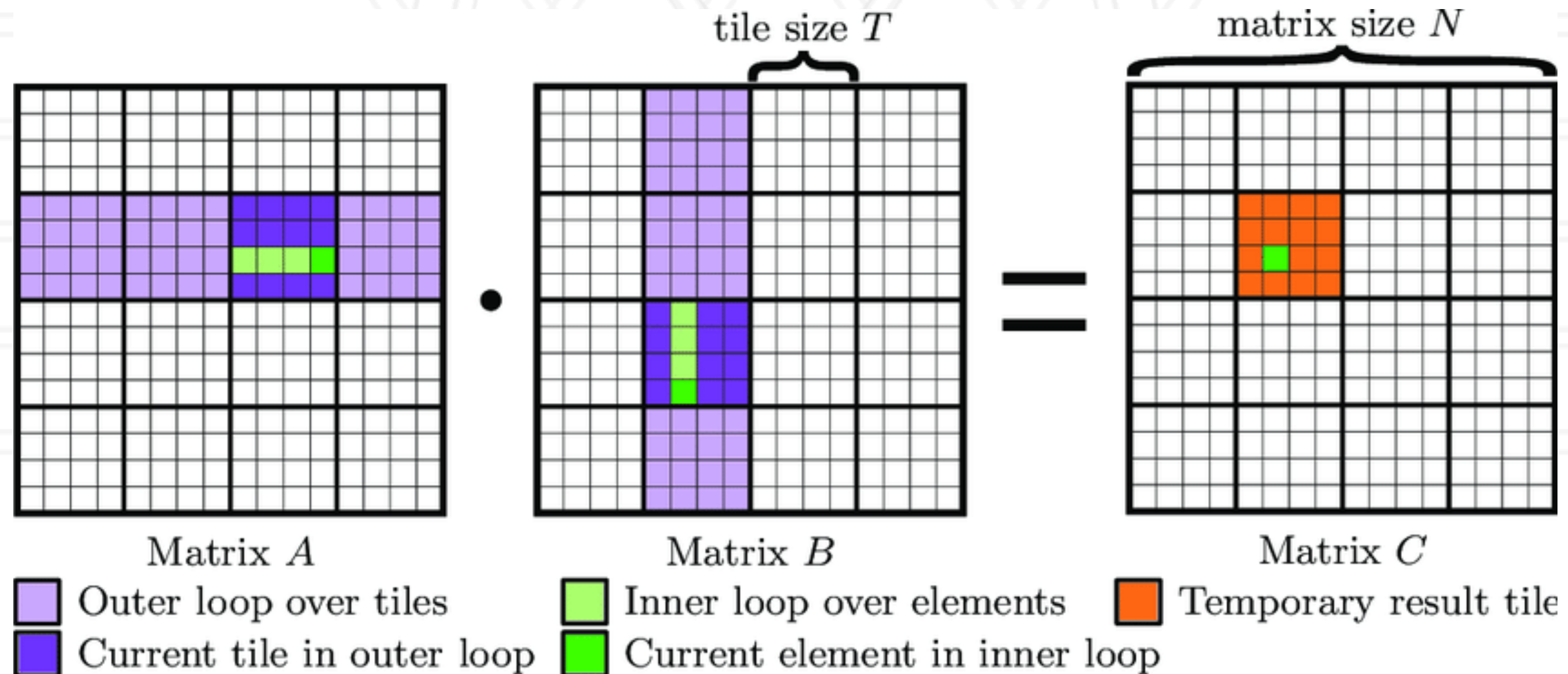DEPARTMENT OF
COMPUTER SCIENCE

# Common pattern in kernels

- Copy from global to shared memory

- `__syncthreads()`: synchronizes all threads in a block

- Perform computation in shared memory

  - `__syncthreads()` as needed

- Copy from shared to global memory

# Reverse array contents using shared memory

```
__global__ void reverse(int *vec) {
  __shared__ int sharedVec[N];

  int idx = threadIdx.x;
  int idxReversed = N - idx - 1;

  sharedVec[idx] = vec[idx];
  __syncthreads();
  vec[idx] = sharedVec[idxReversed];
}
```

DEPARTMENT OF
COMPUTER SCIENCE

# Matrix multiply using shared memory

- Each block $(i, j)$ computes a sub-block $C_{ij}$

- Individual threads in the block work on different elements of sub-block $C_{ij}$



tile size $T$

matrix size $N$

Matrix $A$ · Matrix $B$ = Matrix $C$

■ Outer loop over tiles ■ Inner loop over elements ■ Temporary result tile
■ Current tile in outer loop ■ Current element in inner loop

# CUDA libraries

- Linear algebra

  - cuBLAS, CUTLASS, cuSPARSE

- Deep Learning

  - cuDNN

- Graphics

  - OpenCV, OpenGL, FFmpeg

DEPARTMENT OF
COMPUTER SCIENCE