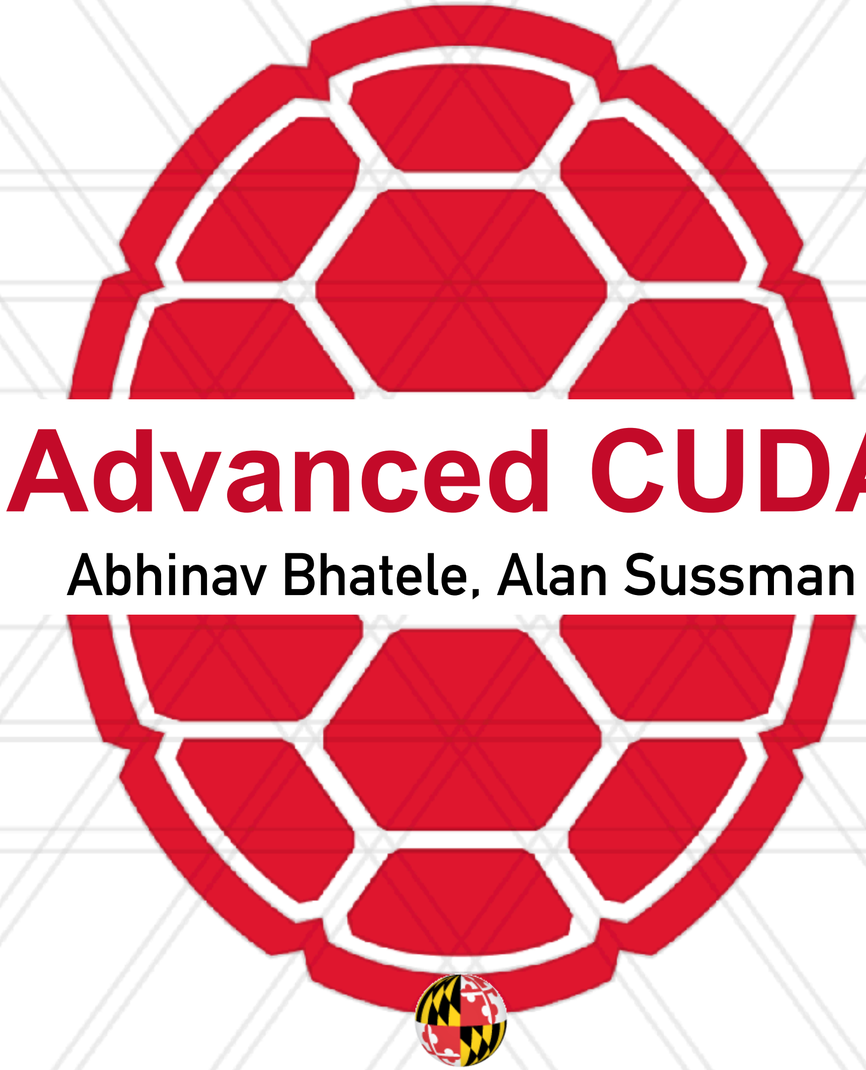
Introduction to Parallel Computing (CMSC416 / CMSC616)



Advanced CUDA

Abhinav Bhatele, Alan Sussman

UNIVERSITY OF MARYLAND

Many slides borrowed from Daniel Nichols' slides

Announcements

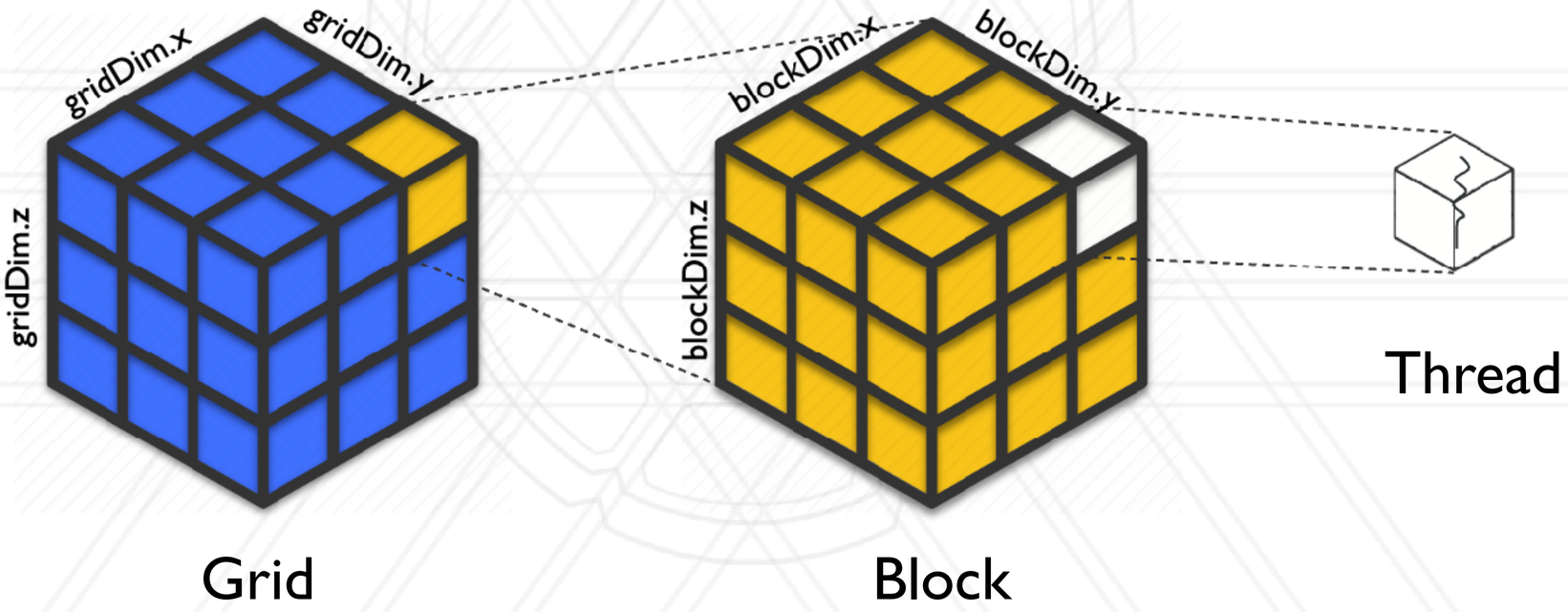
- Assignment 3 is due on October 18 11:59 pm
- Assignment 4 will be posted after the midterm on Oct 28
- Midterm on October 24 during class

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

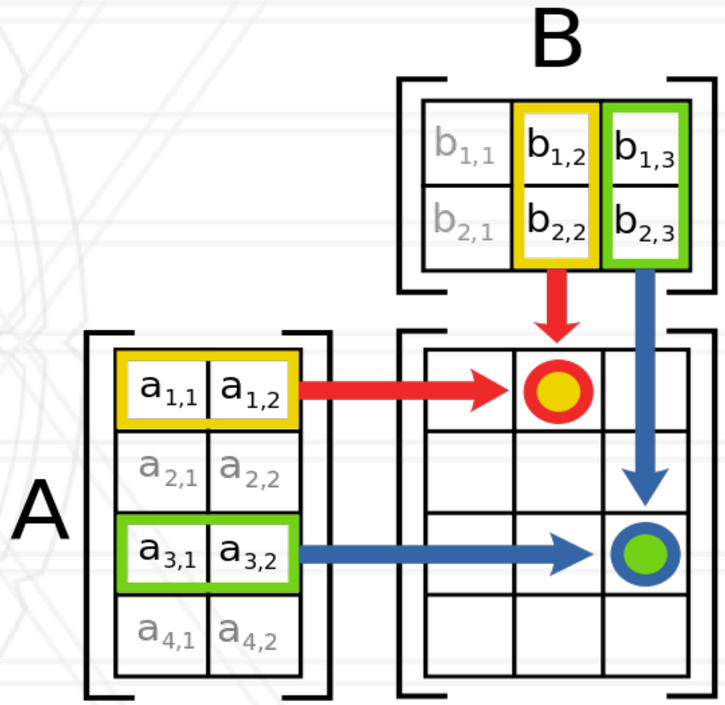


Adding 2D matrices

```
__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

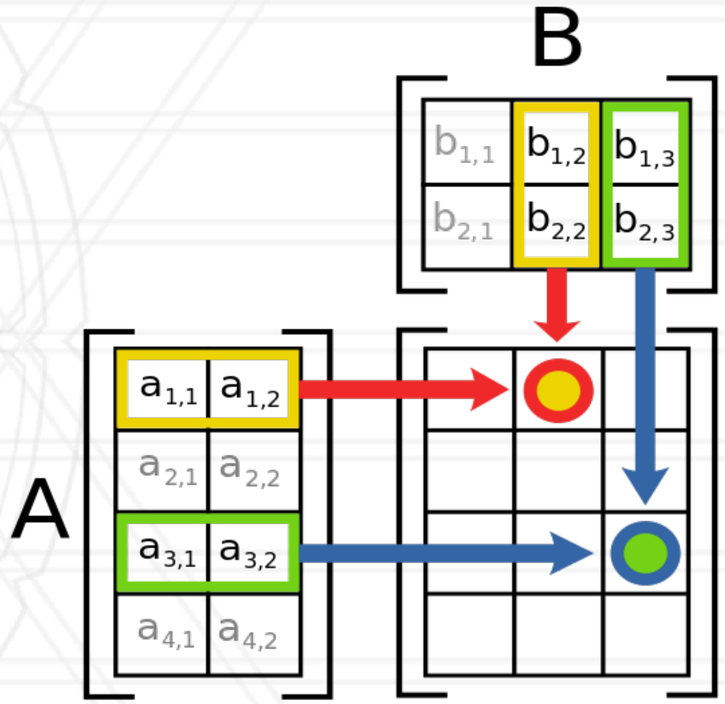
```
for (i=0; i<M; i++)
  for (j=0; j<N; j++)
    for (k=0; k<L; k++)
      C[i][j] += A[i][k]*B[k][j];
```



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

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}



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

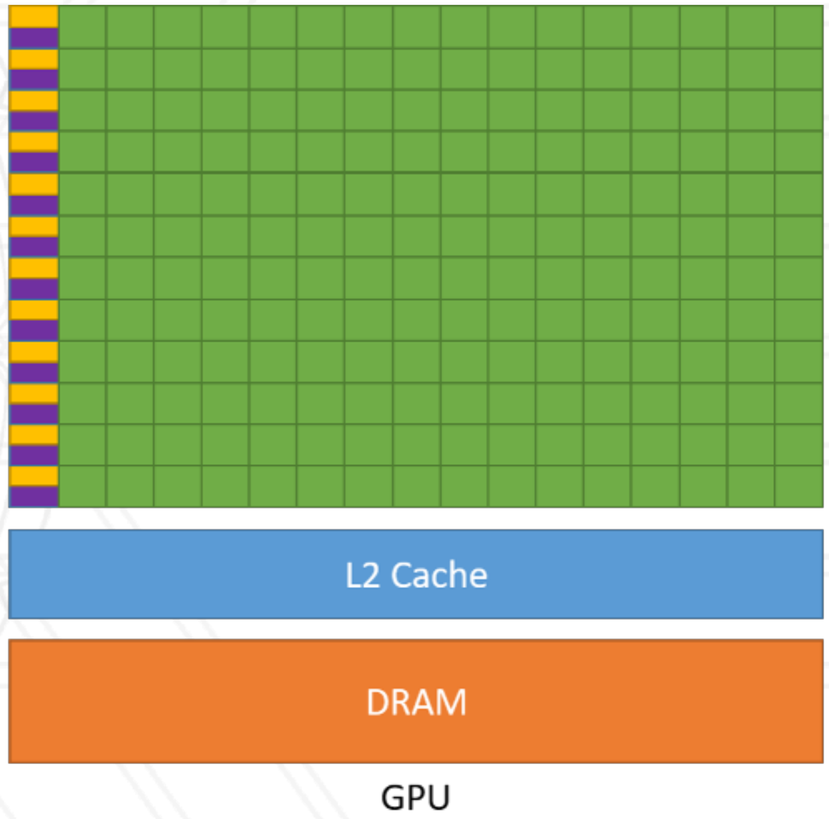
```
int main() {
    dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlocks(M/threadsPerBlock.x + (M % threadsPerBlock.x != 0),
                   N/threadsPerBlock.y + (N % threadsPerBlock.y != 0));

    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];
        }
    }
}
```

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



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

Different types of GPU memory

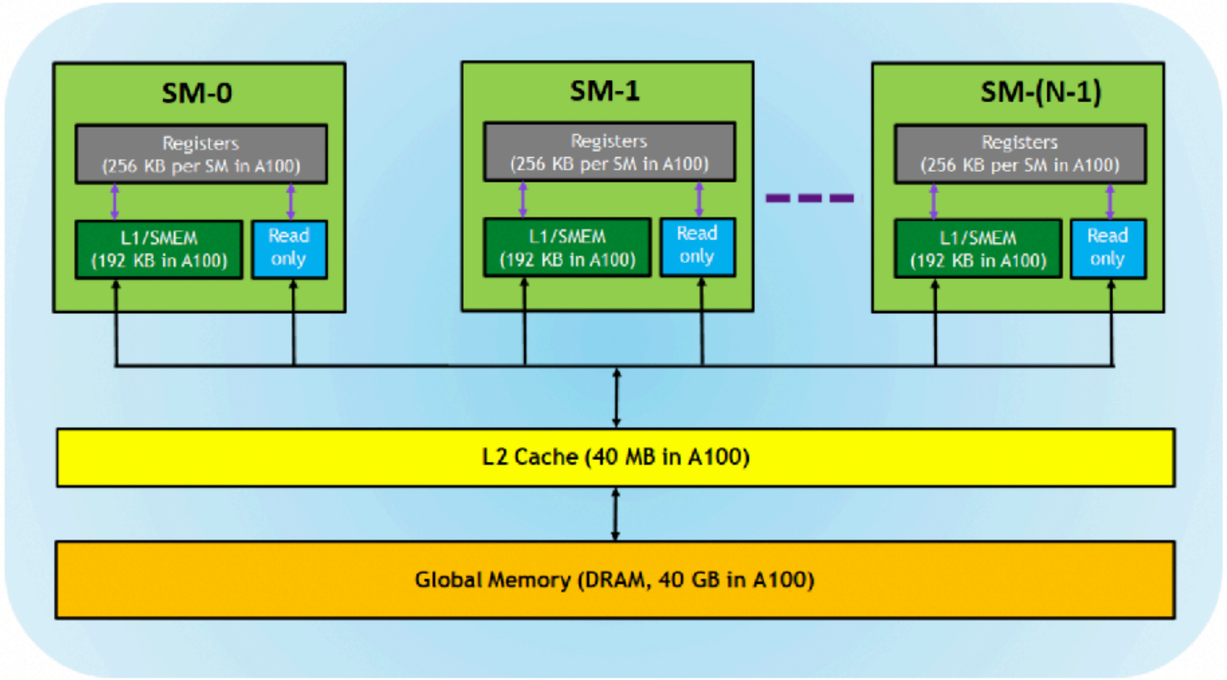
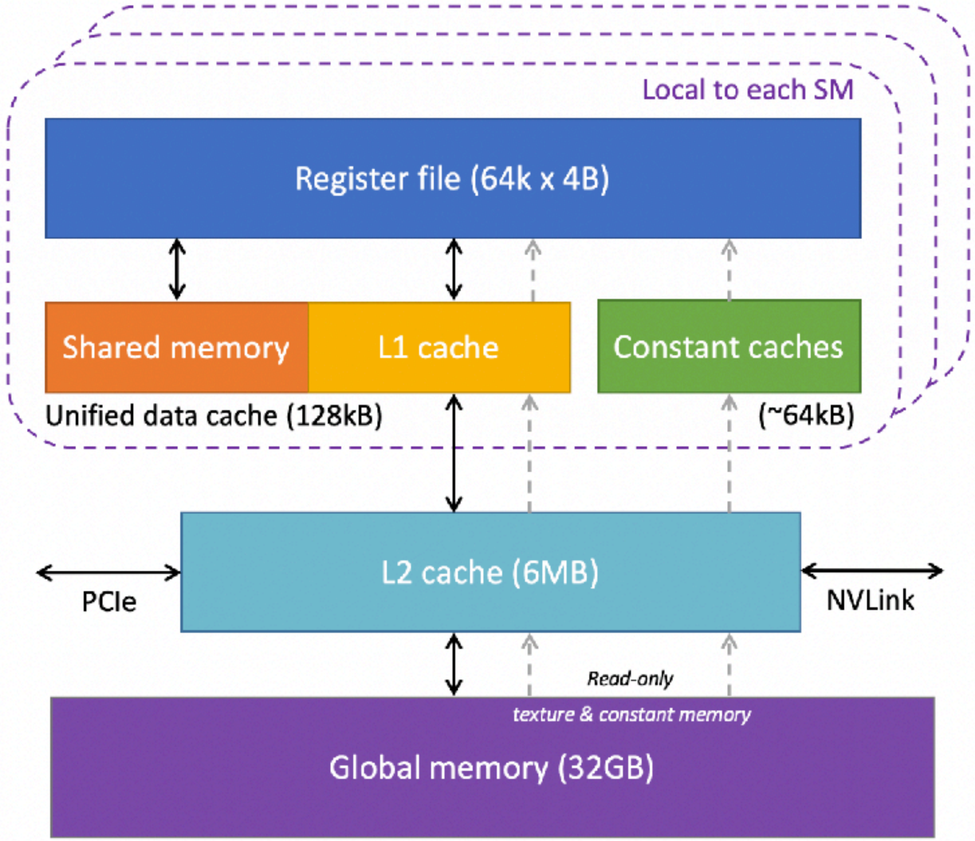


Figure 4. Memory hierarchy in GPUs.



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

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

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

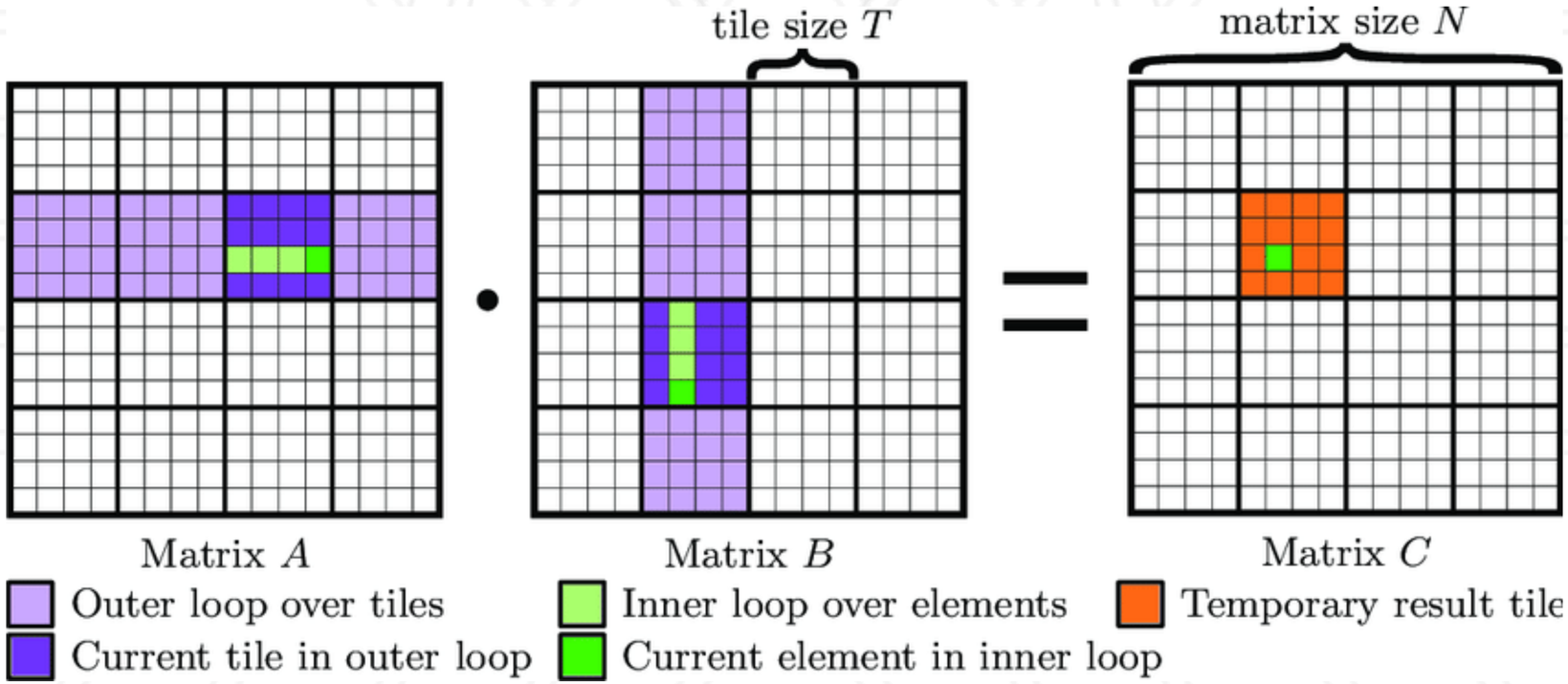
```
__global__ void reverse(int *vec) {
    __shared__ int sharedVec[N];

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

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

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}

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}



Unified memory

- Data is allocated both on the CPU and GPU
- GPU takes care of the synchronization

```
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);

    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();

    … use data on CPU …
    cudaFree(data);
}
```

CUDA libraries

- Linear algebra
  - cuBLAS, CUTLASS, cuSPARSE
- Deep Learning
  - cuDNN
- Graphics
  - OpenCV, OpenGL, FFmpeg

Profiling GPUs

- HPCToolkit + Hatchet
  - Add -e gpu=nvidia to the hpcrun command line
- Nsight Systems
  - Whole application tracing
- Nsight Compute
  - Kernel-level profiling

https://docs.nvidia.com/nsight-systems/UserGuide/index.html

Nsight Systems

- Outputs a .qdrep file that can be visualized in the Nsight GUI

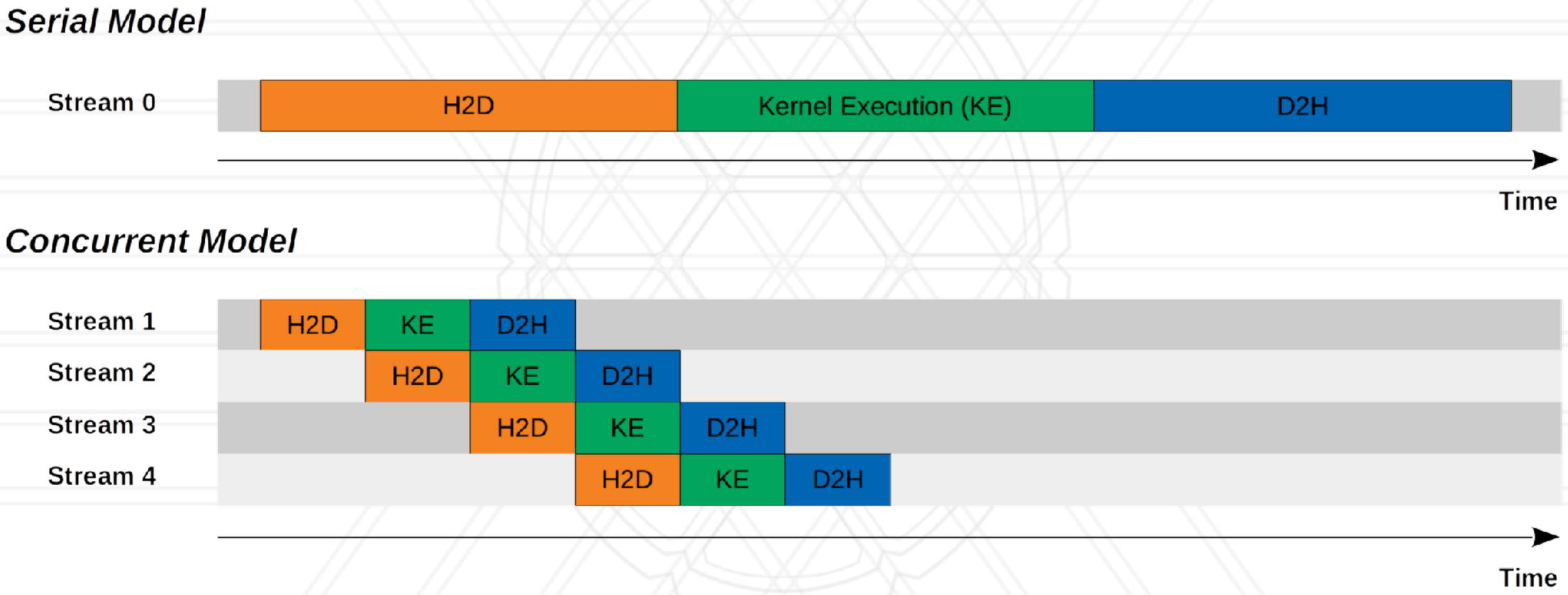
```
nsys profile -t {cuda, nvtx, openmp, mpi, ...}
```

CUDA streams

- Streams can be used to perform multiple CUDA operations simultaneously
  - CUDA Kernel <<<>>>
  - cudaMemcpyAsync()
  - Operations on the CPI

```
int main() {
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    ...

    kernel<<<grid, block, 0, stream>>>(x, b);
}
```

Serial vs. concurrent execution in streams



Launching all streams at once

```
cudaStream_t stream[nStreams];
for (int i = 0; i < nStreams; i++) {
   cudaStreamCreate(&stream[i]);
}

for (int i = 0; i < nStreams; i++){
   int offset = i * streamSize;
   cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes,
                                   cudaMemcpyHostToDevice, stream[i]);

   kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a,
offset);

   cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes,
                                   cudaMemcpyDeviceToHost, stream[i]);
}
```

UNIVERSITY OF
MARYLAND