CUDA

- Software ecosystem for NVIDIA GPUs
- Language for programming GPUs
  - C++ language extension
  - *.cu files
- NVCC compiler

```bash
> nvcc -o saxpy --generate-code arch=compute_80,code=sm_80 saxpy.cu
> ./saxpy
```
CUDA Syntax

```c
__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);
    ...
}
```
Possible Issues?

```c
__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);
    ...
}
```
Possible Issues?

```c
__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:
- \( N > 1024 \)?
- \( N > \# \text{ device threads} \)?
Multiple Blocks

```c
__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 threadsPerBlock = 512;
int numBlocks = N/threadsPerBlock + (N % threadsPerBlock != 0);
saxpy<<<numBlocks, threadsPerBlock>>>(x, y, alpha, N);
```
__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];
}
Grid and Block Dimensions

- # of blocks and threads per block can be 3-vectors
- Useful for algorithms with 2d & 3d data layouts
Grid and Block Dimensions

GRID

gridDim.x

gridDim.y

gridDim.z

BLOCK

blockDim.x

blockDim.y

blockDim.z

THREAD
Grid and Block Dimensions

```cpp
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);
```
Grid and Block Dimensions

Each block is 16x16 threads.

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);
Grid and Block Dimensions

The grid is \([M/16] \times [N/16]\) blocks.

```c
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);
```
Grid and Block Dimensions

```c
__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];
}
```
Questions?
Matrix Multiply

● Standard matrix multiply
● How can we parallelize?

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

- $C_{ij}$ can be computed independent of other values of $C$
- 2-D thread decomposition
- Thread $(i, j)$ computes $C_{ij}$

Matrix Multiply

- Launch $M \times N$ threads
- Thread $(i,j)$ computes $C_{ij}$

```cpp
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, P, N);
```
Matrix Multiply

```c
__global__ void matmul(double *C, double *A, double *B, size_t M, size_t P, 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 < P; k++) {
            C[i*N+j] += A[i*P+k]*B[k*N+j];
        }
    }
}
```

**Compute C\_{ij}**
Issues?
Issues?

● Poor data re-use
  ○ Every value of A & B is loaded from global memory
Issues?

- Poor data re-use
  - Every value of A & B is loaded from global memory
  - A is read N times
  - B is read M times
Issues?

● Poor data re-use
  ○ Every value of A & B is loaded from global memory
    ○ A is read N times
    ○ B is read M times
● How can we improve data re-use?
Shared Memory

- Local
  - thread only
- Shared
  - threads in block
- Global
  - all threads
Shared Memory

**SM-0**
- Registers (256 KB per SM in A100)
- L1/SMEM (192 KB in A100) Read only

**SM-1**
- Registers (256 KB per SM in A100)
- L1/SMEM (192 KB in A100) Read only

**SM-(N-1)**
- Registers (256 KB per SM in A100)
- L1/SMEM (192 KB in A100) Read only

**L2 Cache (40 MB in A100)**

**Global Memory (DRAM, 40 GB in A100)**
Shared Memory

- __shared__
  - Denotes shared memory
- __syncthreads()__
  - Synchronizes all threads in block
__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];
}
Reversing with Shared Memory

```c
__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];
}
```

Allocate N ints in block.
Reversing with Shared Memory

```c
__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];
}
```

Allocate N ints in block.

Store into shared mem. Synchronize.
Load from shared mem.
Matrix Multiply with Shared Memory

- How can we speed up matrix multiply with shared memory?
Matrix Multiply with Shared Memory

- Data Reuse
  - A is read $N$ times
  - B is read $M$ times
Matrix Multiply with Shared Memory

- Block computation
- Each block computes submatrix of C
- Save reused values in shared memory

Matrix Multiply with Shared Memory

- Compute $C = AB + C$
Matrix Multiply with Shared Memory

- Block \((i, j)\) computes \(C_{ij}\) sub matrix
  - Save \(A\) & \(B\) submatrices into shared memory

![Diagram showing matrices A, B, and C with a block of shared memory highlighted]
Matrix Multiply with Shared Memory

- Block \((i, j)\) computes \(C_{ij}\) submatrix
  - Save \(A\) & \(B\) submatrices into shared memory
  - Accumulate partial dot product into \(C\)
Matrix Multiply with Shared Memory

- Block \((i, j)\) computes \(C_{ij}\) sub matrix
  - Save \(A\) & \(B\) submatrices into shared memory
  - Accumulate partial dot product into \(C\)
Matrix Multiply with Shared Memory

- Block \((i, j)\) computes \(C_{ij}\) sub matrix
  - Save \(A\) & \(B\) submatrices into shared memory
  - Accumulate partial dot product into \(C\)
Matrix Multiply with Shared Memory

- A is read $N / \text{block\_size}$ times
- B is read $M / \text{block\_size}$ times
- Data reads from global memory are reduced by an order of the block size

Reference Implementation: 
https://github.com/NVIDIA/cuda-samples/blob/master/Samples/matrixMul/matrixMul.cu
How much faster is it?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time* (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple CPU</td>
<td>170.898</td>
</tr>
<tr>
<td>Simple GPU</td>
<td>1.997</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>0.091</td>
</tr>
<tr>
<td>CuBLAS</td>
<td>0.017</td>
</tr>
</tbody>
</table>

A, B are 2048x2048
* on DeepThought2
Questions?
Profiling GPUs

- HPCToolkit + Hatchet
  - In addition to normal HPCToolkit commands
    - hpcrun -e gpu=nvidia ...
    - hpcstruct <measurements_dir>
- NSight
  - NVIDIA profiling suite
NSight

- nsys command to profile
  - nsys profile -t cuda <executable> <args>
  - Outputs .qdrep file
- View profile in NSight GUI
  - nsys-ui report1.qdrep

NSight

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SM [%]</td>
<td>0.02</td>
<td>0.39</td>
<td>14.43</td>
<td>0.39</td>
<td>0.34</td>
</tr>
<tr>
<td>Memory [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1/TEX Cache [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating Point Operations Roofline

- **Ridge Point**
- **Peak Performance Boundary**
- **Memory Bandwidth Boundary**
- **Achieved Value**

Image from https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline-charts
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SM [%]</td>
<td>0.02</td>
<td>0.39</td>
<td>14.43</td>
<td>0.39</td>
<td>0.34</td>
</tr>
<tr>
<td>Memory [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1/TEX Cache [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DRAM [%]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Floating Point Operations Roofline

Image from https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#roofline-charts
Streams

- Kernels execute in streams
- Stream is passed to kernel invocation
- Streams can execute concurrently

```c
cudaStream_t stream;
...
kernel<<<grid, block, 0, stream>>>(x, b);
```

More info
Streams

Serial Model

H2D Engine: 0
Kernel Engine: 0
D2H Engine: 0

Time

Image from https://leimao.github.io/blog/CUDA-Stream/
Streams

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

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]);
}

for (int i = 0; i < nStreams; i++) {
    cudaStreamDestroy(stream[i]);
}
Streams

Serial Model

- H2D Engine: 0
- Kernel Engine: 0
- D2H Engine: 0

Concurrent Model

- H2D Engine: 1, 2, 3, 4
- Kernel Engine: 1, 2, 3, 4
- D2H Engine: 1, 2, 3, 4

Image from https://leimao.github.io/blog/CUDA-Stream/
Unified Memory

- Data is on both GPU and CPU
- GPU takes care of synchronization
- Incurs small overhead

```c
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);
}
```

Higher Level GPU Programming
Higher Level GPU Programming

- Linear Algebra
  - CuBLAS, MAGMA, CUTLASS, Eigen, CuSPARSE, ...

Higher Level GPU Programming

- **Linear Algebra**
  - CuBLAS, MAGMA, CUTLASS, Eigen, CuSPARSE, ...

- **Signal Processing**
  - CuFFT, ArrayFire, …
Higher Level GPU Programming

- Linear Algebra
  - CuBLAS, MAGMA, CUTLASS, Eigen, CuSPARSE, ...
- Signal Processing
  - CuFFT, ArrayFire, ...
- Deep Learning
  - CuDNN, TensorRT, ...
Higher Level GPU Programming

- Linear Algebra
  - CuBLAS, MAGMA, CUTLASS, Eigen, CuSPARSE, …
- Signal Processing
  - CuFFT, ArrayFire, …
- Deep Learning
  - CuDNN, TensorRT, …
- Graphics
  - OpenCV, FFmpeg, OpenGL, …
Higher Level GPU Programming

- Linear Algebra
  - CuBLAS, MAGMA, CUTLASS, Eigen, CuSPARSE, …
- Signal Processing
  - CuFFT, ArrayFire, …
- Deep Learning
  - CuDNN, TensorRT, …
- Graphics
  - OpenCV, FFmpeg, OpenGL, …
- Algorithms and Data Structures
  - Thrust, Raja, Kokkos, OpenACC, OpenMP, …

FEARLESS IDEAS
When to use GPUs?
Big Picture

- When to use GPUs?
  - Data parallel tasks & lots of data
  - Performance/$$$ and time-to-solution
When to use GPUs?
- Data parallel tasks & lots of data
- Performance/$$$ and time-to-solution

What software/algorithm to use?
Big Picture

- When to use GPUs?
  - Data parallel tasks & lots of data
  - Performance/$$$ and time-to-solution
- What software/algorithm to use?
  - Performance critical
    - Native languages
  - Development time & maintainability
    - higher level APIs