CUDA

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

```
> 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:
- \( N > 1024? \)
- \( N > \# \text{ device threads} \)?
__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 * blockDim.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

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 \( \lceil \frac{M}{16} \rceil \times \lceil \frac{N}{16} \rceil \) blocks.

```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

```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

__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

- `__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];
}
```
Reversing with 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];
}
```

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

- Block \((i, j)\) computes \(C_{ij}\) submatrix
  - 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?

Compare GPU Algorithms

<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

Streams

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

```
cudaStream_t stream;
...
kernel<<<grid, block, 0, stream>>>(x, b);
```
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/](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, ...
An Example: Raja

RAJA::View<double, RAJA::Layout<DIM>> Aview(A, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Bview(B, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Cview(C, N, N);

RAJA::forall<RAJA::loop_exec>(row_range, [=](int row) {
    RAJA::forall<RAJA::loop_exec>(col_range, [=](int col) {

        double dot = 0.0;
        for (int k = 0; k < N; ++k) {
            dot += Aview(row, k) * Bview(k, col);
        }
        Cview(row, col) = dot;
    });
});

See https://raja.readthedocs.io/en/v0.13.0/tutorial/matrix_multiply.html
An Example: Raja

RAJA::View<double, RAJA::Layout<DIM>> Aview(A, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Bview(B, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Cview(C, N, N);

RAJA::forall<RAJA::loop_exec>( row_range, [=](int row) {
    RAJA::forall<RAJA::loop_exec>( col_range, [=](int col) {
        double dot = 0.0;
        for (int k = 0; k < N; ++k) {
            dot += Aview(row, k) * Bview(k, col);
        }
        Cview(row, col) = dot;
    });
});

See https://raja.readthedocs.io/en/v0.13.0/tutorial/matrix_multiply.html
An Example: Raja

RAJA::View<double, RAJA::Layout<DIM>> Aview(A, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Bview(B, N, N);
RAJA::View<double, RAJA::Layout<DIM>> Cview(C, N, N);

RAJA::forall<RAJA::loop_exec>( row_range, [=](int row) {
    RAJA::forall<RAJA::loop_exec>( col_range, [=](int col) {
        double dot = 0.0;
        for (int k = 0; k < N; ++k) {
            dot += Aview(row, k) * Bview(k, col);
        }
        Cview(row, col) = dot;
    });
});

Kernel Execution Policy
- OpenMP
- CUDA
- AMD GPU
- Serial

See https://raja.readthedocs.io/en/v0.13.0/tutorial/matrix_multiply.html
Big Picture

- When to use GPUs?
Big Picture

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

- 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