Review

Sparse matrix representations
Coordinate form
Compressed Sparse Row (CSR method)
Some sparse matrix algorithms
Matrix * Vector product
Different ways of distributing sparse matrix and vector

Basic introduction to GPUs and Cuda

Logical organization of threads
Working group or Block of threads
All threads execute the same instructions
Several blocks in a grid
Several grids
3D structure of threads and grids

CSCE 5160 Parallel Processing

Project Reports Due on the last day of classes: May 1, 2019
Final Exam: Monday May 6, 2019: 1:30-3:30, F280
Complete Course Evaluations using the new SPOT (Student Perception Of Teaching)

Logical organization of threads

Threads are grouped into blocks
(possibly 16 or 32 or 64 threads per block)

Blocks are grouped into Grids
(can be 8, 16 or 32 blocks per grid)

Grids can be organized in 3D grids

Each thread, block can be 1, 2 3 Dimensional
CSCE 5160 Parallel Processing

Physical organization

A processor = Compute Unit = CU

- Contains 32 threads, share instruction unit
- Each thread operates on a different index

A streaming multiprocessor (SM) contains several CUs \(\rightarrow\) equivalent to a grid

- Threads in a CU share registers and local memory
- Threads in a SM share local (shared) memory
- Global/Device memory \(\rightarrow\) used to send and receive data between CPU and CPU

Other types of memory

- Constant cache: remains between kernel invocations

When a kernel (the thing you define in .cu files) is called, the task is divided up into threads

- Each thread handles a small portion of the given task

The threads are divided into a Grid of Blocks

- Both Grids and Blocks are 3 dimensional
  - e.g.
    - \(\text{dim3 dimBlock}(8, 8, 8)\);
    - \(\text{dim3 dimGrid}(100, 100, 1)\);

Kernel\(\llll<\llll<\llll<\llll<\llll<(\ldots);\)

- However, we’ll often only work with 1 dimensional grids and blocks
  - e.g. Kernel\(\llll<\llll<\llll<\llll<\llll<(\ldots);\)
Maximum number of threads per block is usually restricted (512 or 1024 max) depending on the machine.

Maximum number of blocks per grid is usually 65535.
- If you go over either of these numbers your GPU will just give up or output garbage data.
- Much of GPU programming is dealing with this kind of hardware limitations!
- This limitation also means that your Kernel must compensate for the fact that you may not have enough threads to individually allocate to your data points.

We need to define number of blocks per grid, number of threads per block.

Simple example: Add elements of two vectors

```c
for (i = 0; i < N; i++) { C[i] = A[i] + B[i]; }
```

// Kernel definition
```c
__global__ void VecAdd(float* A, float* B, float* C)
{ int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
```

Each thread performs one addition (one iteration)
The index used by the thread is equal to the thread ID. (Id can be obtained using threadIdx or threadIdx (if a block is defined with 2 dimensions))

The main is invoking N threads, one block per grid and N threads per block.

Compile using `nvcc filename.cu --o outputfile`
Continuing with vector add example

for (i=0; i<N; i++) C[i] = A[i]+B[i];

int main()
{
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors
    ...
    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    // Free host memory
    ...
}

Sample programs on arch are at /usr/local/cuda-10.1/samples/
Test at least the vectorAdd application
Consider addition elements of two matrices

```c
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) C[i][j] = A[i][j] + B[i][j];
}
```

// Kernel definition
```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
```

Here also we are using one block but the number of threads per block are $N \times N \rightarrow 2D$ block

In this case, each thread will be identified by x and y IDs

We can extend this to matrix multiplication

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N], int size)
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = 0.0;
    for (int k = 0; k < size; k++)
        C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
```

// Kernel invocation with one block of $N \times N \times 1$ threads
```c
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```
CSCE 5160 Parallel Processing

Same example but different thread configuration (can be extended to matrix multiplication)

```c
int main()
{
   Number of threads per block is limited (512 or 1024)
   Here we limit to 256 but use more blocks
   ...
   // Kernel invocation
   dim3 threadsPerBlock(16, 16);
   dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

// Kernel definition
```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
      C[i][j] = A[i][j] + B[i][j];
}
```

```c
int main()
{
   ...
   // Kernel invocation
   dim3 threadsPerBlock(16, 16);
   dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
   MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
   ...
}
```

---

CSCE 5160 Parallel Processing

Allocating memory for 2D and 3D arrays. Cuda recommends that we use special Malloc functions that align allocation to meet memory accesses in terms of accessing on 4, 8, 16 byte boundaries

```c
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch, width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);
```

```c
// Device code
__global__ void MyKernel(float* devPtr, 
size_t pitch, int width, int height) 
{
for (int r = 0; r < height; ++r) 
{
   float* row = (float*)((char*)devPtr + r * pitch);
   for (int c = 0; c < width; ++c) {
      float element = row[c];
   }
}
```

Pitch = stride = how many elements per column (second dimension)

```
```
Thread configuration should take advantage of how GPU memory is organized
Threads in a block share memory (and registers)
Blocks share local memory
All blocks have access to global memory

We need to send the data from CPU memory to GPU memory
GPU (Device) memory is accessible to both CPU and GPU
So, we need to first allocate space in Device memory for kernel data
Then move the data from CPU memory to GPU/Device memory
And when kernel completes, move the data from Device memory to CPU memory

Consider the following variation of matrix multiplication, where threads in a block move their A and B matrix portions to shared memory.

Note if a thread is computing C[i][j], it needs i'th row of A and j'th column of B.

So, we will group all the row of A needed by threads in a block and columns of B and move them to shared memory

Each block is responsible for computing a portion of C, a submatrix of C.

The rows of A and columns of B are moved to shared memory.
Chapter 3. Programming Interface

CSCE 5160 Parallel Processing

GetSubMatrix are functions that copies the appropriate rows of A or columns of B

Shared memory declaration here allows threads to share the memory

These functions copy specific element of A and B

→ Each thread copies one element of A and one element of B

→ So, all the necessary row and column elements

Write values to result matrix C

The rest of the code

This is the routine to get an element of a matrix we used in the previous slide

This stores a value in a matrix

This identifies the rows of A needed
And declares a submatrix of that size
Declaring data as "const" allows compiler move the data into "constant cache" on GPU
Typically, they are read only
You can copy the data and then modify

GPU’s don’t have locks but they do have atomic instructions

atomicAdd()
int atomicAdd(int* address, int val);

unsigned int atomicAdd(unsigned int* address, unsigned int val);
float atomicAdd(float* address, float val);

These functions proceed sequentially.
For example, each thread computes one multiplication value =A[i][j]*B[i][j];

We can do atomicAdd(&C[i][j], value)

All threads add their values, but the addition proceeds sequentially

Cuda also has atomic subtract, multiply, min, max, etc
There are many other mechanisms and debugging tools provided by NVIDIA to optimize programs.

Note since GPUs rely on SIMD parallelism → all threads execute the same instruction,

Algorithms with variance for thread code do not perform well

Thread Divergence.

Consider for example, Gaussian Elimination

Remember: Divide and Elimination Steps
Not all threads execute these steps in each iteration

Discrete Optimizations (or search solutions space) – Chapter 11

In general an optimization attempts to satisfy some constraints while minimizing or maximizing some objectives. We can describe the problem as

Maximize (or minimize) \((c^T)x\) subject to \(Ax \geq b\)

Here \(A\) is a \(n \times n\) matrix; \(x\) is a \(n \times 1\) vector of unknowns; \(b\) and \(c\) are \(n \times 1\) vectors.

Consider the example on page 471. We are given the following problem

Minimize \(2x_1 + x_2 - x_3 - 2x_4\) subject to

\[
\begin{align*}
5x_1 + 2x_2 + x_3 + 2x_4 & \geq 8 \\
x_1 - x_2 - x_3 + 2x_4 & \geq 2 \\
3x_1 + x_2 + x_3 + 3x_4 & \geq 5
\end{align*}
\]

We need to try to first solve for the unknowns; but we have only 3 equations with 4 unknowns.

We need to try to find all possible values that meet the constraints; and then find one set of values that minimizes the cost function
We can describe this problem as a graph algorithm also. For now let us assume that each $x_i$ can only take values of 0 and 1. Under this assumption we refer to problems of this type as “satisfiability”.

Let us generate a tree with different values assigned to the various $x_i$ variables.

Initial state

Not feasible since the first constraint will not be satisfied

5$x_1$ + 2$x_2$ + $x_3$ + 2$x_4$ $\geq$ 8

Continuing:

Two feasible solutions: $x_1$ = 1; $x_2$ = 0; $x_3$ = 1; $x_4$ = 1 but the constraint equation becomes 0

Constraint Equation: 2$x_1$ + $x_2$ - $x_3$ - 2$x_4$
Consider another example. See page 470 which describes the 8-puzzle problem.

The goal is to start with the initial configuration and by moving the blank tile, rearrange the remaining tile in numerical order.

At each position, we have up to 4 possible new configurations by moving the blank tile left, right, up or down.

One of these 4 moves can be discarded since it results in the same configuration as the one we started with.

The cost function for such problems is often the **number of moves** needed.

We need to have a way of estimating this cost.

Once again we can think of constructing a graph to describe the possible moves that lead us to a feasible solutions.

In general, we are trying to explore the space of all feasible solutions, evaluate each solution for its cost (or goal).