CMSC416: Introduction to Parallel Computing

Topic: GPGPUs and CUDA
Date: March 26, 2024

- **GPGUs (General Purpose Graphics Processing Unit)**
  - It was originally developed to handle computation related to graphics processing
  - useful for scientific computing
  - It is currently used for AI training, bitcoin mining, and high performance parallel computing, etc.

- **Types of Accelerators**
  - IBM’s Cell processors
    - Used in playstation 3
  - GPUs: NVIDIA, AMD, Intel
  - FPGAs (Field Programmable Gate Arrays)

- **Uses for mainstream High Performance Computing**
  - 2013: NAMD, used for molecular dynamics simulations on a supercomputer with 3000 NVIDIA Tesla GPUs
    - They were able to simulate the Aids virus

- **CPU Hardware**
  - Each core has its own L1 cache
  - The L2 caches are shared across multiple cores
  - The L3 cache shared across all cores

- **GCGPU Hardware**
  - It has Many more cores
  - The L1 caches share multiple cores
  - The L2 cache shares all cores
  - Has higher instruction throughput and hides memory access latency with computation

- **GPU vs CPU**
  - GPU has many more cores
  - CPU has higher Clock Speed(GHz)
    - This is caused by heating

- **Volta GV100**
  - Cuda Core
    - Single serial execution unit
      - Can execute instructions
  - Many cores are divided into Streaming Multiprocessors
  - Streaming Multiprocessor (SM)
    - 64 FP32 cores (single precision)
    - 64 INT32 cores
    - 32 FP64 cores (double precision)
    - 8 Tensor cores
      - Used for matrix multiply
A CUDA capable device or GPU is a Collection of SMs

NVLink - Sends messages very fast between GPUs

CUDA
- Allows developers to use C++ as a high-level programming language
- Built around threads, blocks and grids
- Terminology:
  - Host: CPU
    - Where you start computation
  - Device: GPU
    - This is where you offload computation to
  - CUDA Kernel: a function that gets executed on the GPU
- You have to figure out as programmer which threads should do what

Cuda Software abstraction
- Thread - One serial unit of abstraction
- Block - A Collection of threads
  - Number of threads in block <= 1024
- Kernel Grid - A Collection of blocks
- A Thread is executed in a CUDA core
- A Block of threads is executed by a CUDA SM
- A Grid is executed by the entire GPU

Three steps to writing a CUDA kernel
- Copy input data from host to device memory (CPU to GPU)
- Load the GPU program (kernel) and execute it
- Copy the results back to host memory

Example Code Copying data to GPU and then back

```c
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);
```

- CudaMalloc - Allocates memory on the gpu
- CudaMemcpy - Copies data to and from different places (host to device, device to host, device to device, host to host, default)
- cudaFree - frees memory allocated
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);
    ...
}
```

- **Global** is required
- In this case there is 1 block with N threads
- Parameters for saxpy are Array x, array y, scalar alpha
- The top block of code(saxpy) specifies what happens for a single thread
- Calling “saxpy” in main is called the “kernel call”

- What happens if array has > 1024 elements (A Block has a max of 1024 elements)
  - You will have each thread work on multiple parts of array
- What happens when the size of array < number of threads provided in launch parameter
  - There is an out of bounds error
  - You should put a check inside that checks if the amount of threads is <= size of array
- Compiling code
  - `nvcc -o saxpy --generate-code arch=compute_80,code=sm_80 saxpy.cu`
  - `./saxpy`
  - `saxpy.cu` is the file name
  - This Compiles host and gpu code at same time
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];
}
```

```c
int main() {
    ...
    int threadsPerBlock = 512;
    int numBlocks = N/threadsPerBlock
                    + (N % threadsPerBlock != 0);
    saxpy<<<numBlocks, threadsPerBlock>>>(x, y, alpha, N);
    ...
}
```

Threads per block and numblocks get passed into the kernel call
- Each thread has an Id. threadIdx.x gives the Id of the current thread
- Each block has a block id. BlockId.x gives the Id of the current block
- int i = blockDim.x * blockDim.x * blockDimId.x + threadIdx.x;
  - This line stores the global threadID in i