

ECE 574 – Cluster Computing

Lecture 18

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

11am, Barrows 133

26 March 2024

Announcements

- HW#7 due
- Still working on HW#6 grading
- Project topics were due, should have responded to them
- ECE571 next semester
- Thanks for those attending the Friday talk
- Faculty interview on Monday moved to Wednesday, so have to cancel Wednesday office hours



HW#6 Notes

- Hard to track down all the issues: Common ones:
 - trying to be fancy
 - C loop bounds: if want to operate on 0 to 79 inclusive, want your loop to go from `for(i=0;i<80;i++);`
 - Off by one errors
 - Subtracting off ystart but forgetting you modified ystart to be 1 in first rank
 - Gathering in the wrong direction
 - Doing border adjustments twice



HW#7 Notes

- Really testing your debugging skills. Advice:
 - Write in small chunks, testing along way. Easier than throwing together big mass of code and then giving up when it doesn't work
 - Test with 1 rank and be sure that works before moving onto more ranks
 - Dump intermediate output, be sure sobelx works before worrying about sobely or combine/
 - Print your ranges and make sure they make sense



- “The output looks the same” , but it isn’t. Try flipping between them. There might be binary diff tools to actually show you what’s different, though that’s more difficult if it’s an off-by-one error.
- Try to understand why you are off by one before just adding +1 or -1 to your code
- If it crashes, usually it means you’re going off the edge of a buffer, double and triple check the values that are going into array accesses (or even worse, pointers)
- Some code is tricky, like finding edge conditions on inputs. This is an important thing that happens often



in programming. Coding isn't always cut+paste of ask an AI, someone has to write the original tricky code.



CUDA Programming

- Since 2006
- Compute Unified Device Architecture
- See the NVIDIA “CUDA C Programming Guide”
- Use `nvcc` to compile
- `.cu` files. Note, technically C++ so watch for things like `new`



CUDA Programming background

- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>
- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>



NVIDIA Terminology (CUDA)

- Thread: chunk of code running on GPU.
- Warp: group of thread running at same time in parallel simultaneously
AMD calls this a “wavefront”
- Block: group of threads that need to run
- Grid: a group of thread blocks that need to finish before next can be started



Terminology (cores)

- Confusing. Nvidia would say GTX285 had 240 stream processors; what they mean is 30 cores, 8 SIMD units per core.



Notes from CUDA document

- Designed to be simple
- Three abstractions
 - hierarchy of thread groups
 - shared memories,
 - barrier synchronization
- Threads can be run in any order on any number of cores, the programmer doesn't have to worry about how many cores there are



CUDA Programming

- Heterogeneous programming – there is a host executing a main body of code (a CPU) and it dispatches code to run on a device (a GPU)
- CUDA assumes host and device each have own separate DRAM memory
(newer cards can share address space via VM tricks)
- CUDA C extends C, define C functions “kernels” that are executed N times in parallel by N CUDA threads



CUDA Programming – Host vs Device

- *host* vs *device*
 - host code runs on CPU
 - device code runs on GPU
- Host code compiled by host compiler (gcc), device code by custom NVidia compiler



CUDA Compiling

- `nvcc` – wrapper around `gcc`. global code compiled into PTX (parallel thread execution) ISA
- can code in PTX code directly which is sort of like assembly language. Won't give out *actual* assembly language. Why?
- PTX code is JIT compiled into native by the device driver
- You can control JIT with environment variables
- Various command line options, some do things like



specify the “compute version” (GPU generation) to target

- version compliance – can check version number. New versions support more hardware but sometimes drop old



CUDA Coding

- Device kernel, in `__global__` section
- Only subset of C/C++ supported in the device code
- CUDA C has mix of host and device code. Compiles the global stuff to PTX, compiles the `<<< ... >>>` into code that can launch the GPU code



CUDA Programming – Memory Allocation

- `cudaMalloc()` allocates memory on the device
`cudaMalloc((void **)&dev_a,N*sizeof(int));`
- `cudaFree()` to free when done
- `cudaMemcpy(dev_a,a,N*sizeof(int),
cudaMemcpyHostToDevice);`
- `cudaMemcpy(c,dev_c,N*sizeof(int),
cudaMemcpyDeviceToHost);`



CUDA Programming – Pointers

- Note: result of a `cudaMalloc()` might look like a pointer, but it's not
- You can't dereference memory allocated with `cudaMalloc()` on the CPU, the memory area is completely separate
- There is work on newer GPUs allowing unified CPU/GPU memory but we're going to assume that's not available



CUDA Programming – 2D/3D Arrays

- Can use `cudaMallocPitch()` and `cudaMalloc3D()`
- These will do proper padding/alignment when using 2d/3d arrays
- Have own `cudaMemcpy2D()` and `cudaMemcpy3D()`



CUDA Hardware – this might be dated

- GPU is array of Streaming Multiprocessors (SMs)
- Program partitioned into blocks of threads. Blocks execute independently from each other.
- Manages/Schedules/Executes threads in groups of 32(??) parallel threads (warps) (weaving terminology) (no relation)
- Threads have own PC, registers, etc, and can execute independently
- When SM given thread block, partitions to warps and



each warp gets scheduled

- One common instruction at a time. If diverge in control flow, each way executed and thread not taking that path just waits.
- Full context stored with each warp; if warp is not ready (waiting for memory) then it may be stopped and another warp that's ready can be run



CUDA Threads

- kernel defined using `__global__` declaration. When called use `<<<...>>>` to specify number of threads
- each thread that is called is assigned a unique ThreadID
Use `threadIdx` to find what thread you are and act accordingly



CUDA Programming – Overview

- Special kernel `__global__` function that runs on GPU
limited what you can run there
- Special call with angle brackets to run in parallel

```
VecAdd<<<1,N>>>(A,B,C)
```

- The kernel is run simultaneously on N different threads
- To get data on GPU need to `cudaMalloc()` it and then `cudaMemcpy()` there
- When done, need to `cudaMemcpy()` back



Simple CUDA Kernel Example

```
__global__ void VecAdd(float *A, float *B, float *C) {  
    int i = threadIdx.x;  
  
    if (i<N)          // don't execute out of bounds  
        C[i]=A[i]+B[i];  
}  
  
int main(int argc, char **argv) {  
    ....  
    /* Invoke N threads */  
    VecAdd<<<1,N>>>(A,B,C);  
}
```



CUDA Programming – Thread Hierarchy

- `threadIdx` – 3 component vector, can identify what index our thread is executing
- one dimensional (x) – thread id is (x)
- two dimensional (x,y) – thread id is $(x + y * xsize)$
- three dimensional (x,y,z) – thread id is $(x + y * xsize + z * xsize * ysize)$



CUDA Programming – 2x2 Example

```
__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];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```



CUDA Example – multidimensional

- threadIdx is 3-component vector, can be seen as 1, 2 or 3 dimensional block of threads (thread block)
- Much like our sobel code, can look as 1D (just x), 2D, (thread iD is $((y * \text{xsize}) + x)$ or $(z * \text{xsize} * \text{ysize}) + y * \text{xsize} + x$)
- Weird syntax for doing 2 or 3d.

```
__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];
}

int numBlocks=1;
```



```
dim3 threadsPerBlock(N,N);  
MatAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);
```

- Each block made up of the threads. Can have multiple levels of blocks too, can get block number with blockIdx
- Thread blocks operate independently, in any order. That way can be scheduled across arbitrary number of cores (depends how fancy your GPU is)



CUDA Programming – Threads

- `__global__` parameters to function – means pass to CUDA compiler
- call global function like this `add<<<1,1>>>(args)`
where first inside brackets is number of blocks, second is threads per block
- Can get block number with `blockIdx.x` and thread index with `threadIdx.x`
- Could have 65536 blocks and 1024 threads, possibly 2 billion blocks on recent hardware.



- Why thread limit? Limited by number of threads per core that share the same memory resources.
- Why threads vs blocks?
Shared memory, block specific
`__shared__` to specify



CUDA Programming – What if too big

- For example, sobel of 320x320x3 size is bigger than 1024 elements
- Need to break up into smaller chunks. This is tricky.
- ```
// Kernel invocation
dim3 threadsPerBlock(16, 16);
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```
- Blocks must be able to operate independently.



# CUDA Programming – Barriers

- `__syncthreads()` is a barrier to make sure all threads finish before continuing





# CUDA Programming – Thread Block Clusters

- On newer GPUs can also have clusters of compute cores that are close together, and you can set up clusters of thread blocks to run on them.



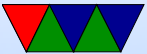
# CUDA Memory

- Per-thread private local memory
- Shared memory (`__shared__`) visible to whole block (lifetime of block)  
Is like a scratchpad, also faster
- Global memory
- also constant and texture spaces. Have special rules.  
Texture can do some filtering and stuff
- Global, constant, and texture persistent across kernel launches by same app.



# CUDA L2 Cache

- Can configure in complex way
- “set aside” parts of the cache



# More Coding – Initialization

- Traditionally no explicit initialization, done automatically first time you do something (keep in mind if timing things)
- With CUDA 12.0 can call `cudaInitDevice()` or `cudaSetDevice()` to force initialization



# More Coding – Global Memory

- Global Memory: linear or arrays.
  - Arrays are textures
  - Linear arrays are allocated with `cudaMalloc()`, `cudaFree()`
  - To transfer use `cudaMemcpy()`
  - Also can be allocated `cudaMallocPitch()` `cudaMalloc3D()` for alignment reasons  
can have better performance
  - Access by symbol (?)



# CUDA Shared memory

- `__shared__`. Faster than Global also `__device__`  
Manually break your problem into smaller sizes
- Example where they do a matrix multiply and copy from global to shared memory for faster work



# Other Memory Interfaces

- Can lock host memory with `cudaHostAlloc()`
- Pinned, can't be paged out. Can load store while kernel running if case.
- Only so much available.
- Can be marked writecombining. Not cached.
- Slow for host to read (should only write) but speeds up PCI transaction.



# Heterogeneous Execution

- Usually assumed that serial code running on CPU while launching the parallel code on GPU





# Async Concurrent Execution

- Instead of serial/parallel/serial/parallel model
- Want to have CUDA running and host at same time, or with mem transfers at same time
  - Concurrent host/device: calls are async and return to host before device done
  - Concurrent kernel execution: newer devices can run multiple kernels at once. Problem if use lots of memory
  - Overlap of Data Transfer and Kernel execution
  - Streams: sequence of commands that execute in order,



but can be interleaved with other streams  
complicated way to set them up. Synchronization and  
callbacks



# Events

- Can create performance events to monitor timing
- PAPI can read out performance counters on some boards
- Often it's for a full synchronous stream, can't get values mid-operation
- NVML can measure power and temp on some boards?



# Multi-device system

- Can switch between active device
- More advanced systems can access each others device memory



# Other features

- Unified virtual address space (64 bit machines)
- Interprocess communication
- Can share device memory handles between processes with IPC



# Error Checking

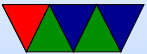
- Complex, as things running asynchronously on GPU
- Various functions to query the error state
- `cudaPeekAtLastError()` – reports error
- `cudaGetLastError()` – resets to `cudaSuccess`

```
// check for error
cudaError_t error = cudaGetLastError();
if (error != cudaSuccess) {
 printf("CUDA error: %s\n", cudaGetErrorString(error));
 exit(-1);
}
```



# Texture Memory

- Complex



# 3D Interop

- Can make results go to an OpenGL or Direct3D buffer
- Can then use CUDA results in your graphics program





# Code Example – see vector\_add.cu

```
#include <stdio.h>

#define N 10

__global__ void add (int *a, int *b, int *c) {
 int tid=blockIdx.x;

 if (tid<N) {
 c[tid]=a[tid]+b[tid];
 }
}

int main(int argc, char **argv) {

 int a[N],b[N],c[N];
 int *dev_a,*dev_b,*dev_c;
 int i;

 /* Allocate memory on GPU */
```



```

cudaMalloc((void **)&dev_a,N*sizeof(int));
cudaMalloc((void **)&dev_b,N*sizeof(int));
cudaMalloc((void **)&dev_c,N*sizeof(int));

/* Fill the host arrays with values */
for(i=0;i<N;i++) {
 a[i]=-i;
 b[i]=i*i;
}

cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);

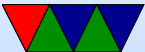
add<<<N,1>>>(dev_a,dev_b,dev_c);

cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);

/* results */
for(i=0;i<N;i++) {
 printf("%d+%d=%d\n",a[i],b[i],c[i]);
}

cudaFree(dev_a);
cudaFree(dev_b);

```



```
 cudaFree(dev_c);

 return 0;
}
```



# Code Examples – saxpy

- Does a single-precision  $A*X+Y$

```
y[i]=a*x[i]+y[i]
```

- `saxpy_c.c` shows the algorithm in C, does 8 million iterations
- `saxpy.cu` does same thing on GPU. Fails for 8 million threads though. Why? Thread can't be higher than 1024
- `saxpy_block.cu` breaks things up into blocks and thus can run. gives same results



- Try timing things with `time`, notice GPU code is actually slower. This is due to the memory copying overhead, if you use the `loop` option to make it repeat eventually hit a crossover point.



# CUDA Tools

- `nvidia-smi`. Various options. Usage, power usage, etc.
- `nvprof ./hello_world` profiling
- `nvvp` visual profiler, can't run over text console
- `nvidia-smi --query-gpu=utilization.gpu,power.draw --format=csv -lms 100`



# CUDA Debugging

- Can download special cuda-gdb from NVIDIA
- Plain printf debugging doesn't really work



# Performance

- Really optimized for 32-bit (single-precision) float
- We will do 32-bit integer, which it also can do
- Intrinsics for faster divide
- Use single-precision `sinf()`, `sqrtf()` and such
- Control flow can really hurt performance, lead to serialization





# C++

- Can do most of C++ to varying degree
- If you want to do advanced C++ stuff check out the CUDA C document for details



# Homework Tips

- First implement combine, as it's simpler
  - You will need to `cudaMalloc()` room for `sobelx`, `sobely`, and result on the device
  - You will need to `cudaMemcpy` the `sobelx` and `sobely` data there
  - Assuming your combine code is already acting as a linear array, converting it to a kernel should be straightforward
  - You will need to split it up into blocks though as the



- image size is too big to fit in one block of threads
- Remember to copy the results back at the end
  - Next implement convolution
    - You probably want to collapse all the loops down to one
    - The hardest issue is skipping the edges. Instead of skipping  $y==0$  and  $y==ysize-1$  you'll need to skip first  $xsize*depth$  and last  $xsize*depth$  chunks, as well as the left/right sides which are something like  $i%(xsize*depth)<3$  and  $i%(xsize*depth)>(xsize*depth-4)$
  - Debugging if things go wrong can be tricky

