ECE 574 – Cluster Computing
Lecture 18

Vince Weaver

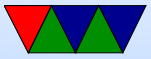
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

28 March 2023

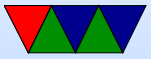
Announcements

- HW#8 (CUDA) will be posted.
- Project topics were responded to.
- ECE598/ECE531 next semester

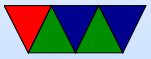


HW#6 Notes

- Graded. If still stuck I have a semi-solution I can send
- Took a long time to figure out what some of the issues were
  - trying to be fancy
  - C loop bounds: if want to operate on 0 to 79 inclusive, want your loop to go from i=0;i¡80;i++
  - Off by one errors
  - Subtracting off ystart but forgetting you modified ystart to be 1 in first rank

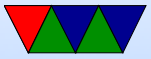


- Academic Honesty



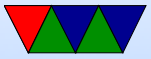
CUDA Programming Link

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



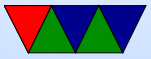
Notes from CUDA document

- TODO: merge this back into lecture 17
- 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



Notes from CUDA document – Programming Model

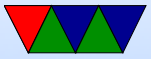
-



CUDA Programming – Review

- 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 simultaenously on N different threads
- To get data on GPU need to cudaMalloc() it and then cudaMemcpy() there
- When done, need to cudaMemcpy() back

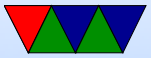


CUDA Example

```
__global__ void VecAdd(float *A, float *B, float *C) {
    int i = threadIdx.x;

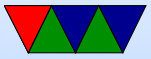
    if (i<N)          // don't execute out of bounds
        C[i]=A[i]+B[i];
}

int main(int argc, char **argv) {
    ....
    /* Invoke N threads */
    VecAdd<<<1,N>>>(A,B,C);
}
```



CUDA Programming – Thread Hierarchy

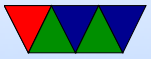
- threadIdx – 3 component vector, can identify what index our thread is executing
- one dimensional (x) – thread id is (x)
- two dimensional (x,y) – thread id is (x +y*xsize)
- three dimensional (x,y,z) – thread id is (x +y*xsize + z*xsize*ysize)



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);
    ...
}
```

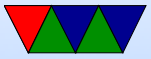


CUDA Example – multidimensional

- 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*xsize)+x) or (z*xsize*ysize)+y*xsize+x
- Weird syntax for doing 2 or 3d.

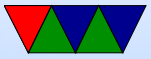
```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i=threadIdx.x;
    int j=threadIdx.y;
    C[i][j]=A[i][j]+B[i][j];
}

int numBlocks=1;
```



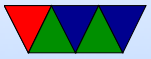
```
dim3 threadsPerBlock(N,N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);
```

- 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)

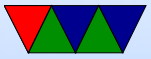


CUDA Programming – Threads

- `__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
- Can have 65536 blocks and 1024 threads (on current hardware?)

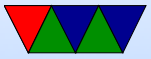


- Why thread limit? Limited by number of threads per core that share the same memory resources.
- Why threads vs blocks?
  Shared memory, block specific
  `__shared__` to specify



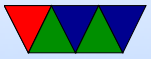
CUDA Programming – What if too big

- 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.


  ```

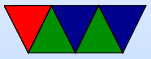
CUDA Programming – Barriers

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



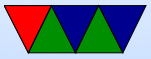
CUDA Programming – Thread Block Clusters

- 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.



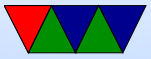
CUDA Memory

- Per-thread private local memory
- Shared memory 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.



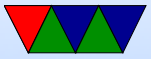
More Coding

- No explicit initialization, done automatically first time you do something (keep in mind if timing)
- 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



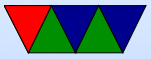
can have better performance

- Access by symbol (?)



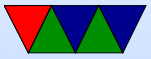
CUDA Shared memory

- `__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



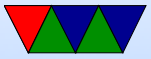
Misc

- 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. So slow for host to read (should only write) but speeds up PCI transaction.



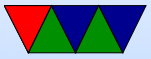
Heterogeneous Execution

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



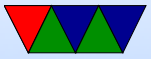
Async Concurrent Execution

- 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,



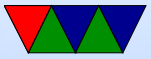
but can be interleaved with other streams

complicated way to set them up. Synchronization and callbacks



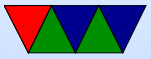
Events

- 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?



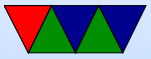
Multi-device system

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



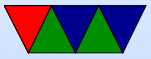
Other features

- Unified virtual address space (64 bit machines)
- Interprocess communication



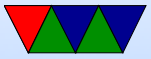
Error Checking

- Complex, as things running asyncronously on GPU
- Various functions to query the error state



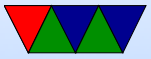
Texture Memory

- Complex



3D Interop

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



Code Example

```
#include <stdio.h>

#define N 10

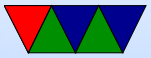

__global__ void add (int *a, int *b, int *c) {
    int tid=blockIdx.x;

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


int main(int arc, 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;
}

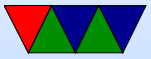
cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);

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

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

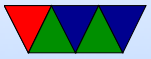
/* results */
for(i=0;i<N;i++) {
    printf("%d+%d=%d\n",a[i],b[i],c[i]);
}

cudaFree(dev_a);
cudaFree(dev_b);
```



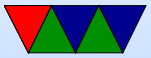
```
    cudaFree(dev_c);

    return 0;
}
```



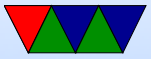
Code Examples

- Go through examples
- Also show off `nvidia-smi`



CUDA Notes

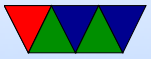
- Nicely, we can use only block/thread for our results, even on biggest files
- In past there was a limit of 64k blocks with "compute version 2" but on "compute version 3" we can have up to 2 billion



CUDA Examples

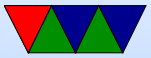
- Make builds them. .cu file, built with nvcc
- `./hello_world` bit of a silly example
- `saxpy.c`
  single a*x+y

  CPU GPU run 320000000 1.12s 2.06
- What happen if thread count too high? Max threads per block 512 on Compute 2, 1024 on compute 3 Above



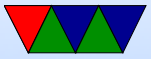
1024? Try saxpy_block

- maximum block size 64k on Compute version 2, 2GB on Compute Version 3 200,000 50,000 cpu = 4.5s gpu = 0.8s



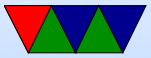
CUDA Tools

- `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.dr`
  `--format=csv -lms 100`



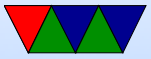
CUDA Debugging

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



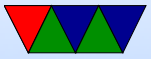
Performance

- 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



Go through HW stuff