ECE 574 – Cluster Computing
Lecture 26

Vince Weaver

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

vincent.weaver@maine.edu

9 November 2026

Announcements

- HW#9 is posted
- Reponded to project topics, if you didn't send them yet, please do
- 11 groups for projects? No clusters this year.
- Still working on HW#6 grading
- Don't forget no class Wednesday (Veteran's Day)

CUDA Programming – Reminder

- Special kernel `__global__` function that runs on GPU limited what you can run there
- Only subset of C/C++ supported in the device code, compiled to PTX
- 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

CUDA Programming – Multidimensional

- For
  the number of blocks and number of threads in a call like `MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);`
  you can specify 1D, 2D or 3D arrangements of blocks and threads
- You declare them something like this, for example a 2D NxN thread array: `dim3 threadsPerBlock(N, N);`
- Inside the kernel you can get your position with threadIdx.x, threadIdx.y, threadIdx.z
- Same with blocks, just blockIdx.x, blockIdx.y, blockIdx.z

CUDA Programming – Undoing Multidimensior

- If you are multidimensional but want to convert back to linear:
  - 1D (x) – thread id is (x)
  - 2D (x,y) – thread id is (x +y*xsize)
  - 3D (x,y,z) – thread id is (x +y*xsize + z*xsize*ysize)
  - blockDim gives the number of dimensions of threads
  - gridDim gives the number of dimensions of blocks

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);
    ...
}
```

Thread/Blocksize Notes Notes

- Number of threads and blocks available depends on your GPU "compute" version
- Before version 3 you could have only 64k blocks per 1D grid, but that has been bumped up to 2 billion
- Maximum threads per block 1024
- Compute Version: Haswell-ep has Pascal (6), Ampere is 8, Blackwell is 9
- Why thread limit? Limited by number of threads per core that share the same memory resources.

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

CUDA Tools

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

CUDA PAPI

- PAPI has both CUDA and NVML components
- Show `papi_native_avail` and `papi_component_avail` results

Cuda C++ Support

- Can do most of C++ to varying degree
- If you want to do advanced C++ stuff check out the CUDA C document for details
- One big one is no exceptions

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];
        }
}
```

Code Example – see vector_add.cu

```
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;
        }
```

Code Example – see vector_add.cu

```
        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. Why? Thread can't be higher than 1024
- `saxpy_block.cu` breaks things up into blocks and thus can run. gives same results
- Uses `<<<(N+255)/256,256>>>` to split up into groups of 256 threads. The first part adds 255 because otherwise if not multiple of 256 you'd not have an extra block with the remainder

Code Examples – saxpy perf

- 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 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

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

Homework #9 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

Homework #9 Tips – 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)`

Homework #9 Tips – Debugging

- Debugging if things go wrong can be tricky
- Some notes from homework. Send all white to see if you get that result. Send the image and get it back
- If things work fine on butterfinger but not larger often it means you are going out of bound, essentially segfaulting the GPU (butterfinger is small enough that it's harder to trigger)