CS 5220

Accelerators

David Bindel

2024-10-10

Logistics

Big picture

- Less low-level stuff after break
- Time to start thinking about final projects!
- Mid-term feedback emails would be useful

Travel impacts

- Next week: Guidi lectures
- Mon, Nov 4: Bindel away, no OH
- Week of Nov 11: Bindel away, guest lectures

Upcoming work

- P2 will be due Oct 22
    - NB: Perlmutter maintenance 10/16
- P3 will be released Oct 22
- HW to be released tonight
- Final project proposal due Oct 29

Balls, bins, and buckets

To the board!

CUDA vecAdd

Basic model

CPU code calls GPU *kernels*

1. First, allocate memory on GPU
2. Copy data to GPU
3. Execute GPU program
4. Wait for completion
5. Copy results back to CPU

Vector addition

"Hello world": vector addition in CUDA

```
#include <iostream>
#include <vector>
#include <cassert>

// Compute y += x
void add(const std::vector<float>& x, std::vector<float>& y);

int main()
{
    int N = 1<<20;
    std::vector<float> x(N), y(N);
    for (auto& xi : x) xi = 1.0f;
    for (auto& yi : y) yi = 2.0f;
    add(x, y);
```

CUDA kernel (v1)

```
__global__
void gpu_add(int n, float* x, float* y)
{
    for (int i = 0; i < n; ++i)
        y[i] += x[i];
}

gpu_add<<<1,1>>>(n, d_x, d_y);
```

- Executes on the GPU with *no parallelism*

CUDA kernel (v2)

```
__global__
void gpu_add(int n, float* x, float* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n)
        y[i] += x[i]
}

// Should have n <= n_blocks * block_size
gpu_add<<<n_blocks, block_size>>>(n, d_x, d_y);
```

- This is a lot of threads...

Full code

```
void add(const std::vector<float>& x, std::vector<float>& y)
{
    int n = x.size();
    int size = n * sizeof(float);
    float *d_x, *d_y;
    cudaMalloc((void**)& d_x, size); cudaMalloc((void**)& d_y
    cudaMemcpy(d_x, x.data(), size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y.data(), size, cudaMemcpyHostToDevice);
    int bs = 256;
    int nblocks = (n+bs-1)/bs;
    gpu_add<<nblocks,bs>>>(n, d_x, d_y);
    cudaMemcpy(y.data(), d_y, size, cudaMemcpyDeviceToHost);
    cudaFree(d_x); cudaFree(d_y);
}
```

CUDA kernel (v3)

```
__global__
void gpu_add(int n, float* x, float* y)
{
    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;
    for (int i = index; i < n; i += stride)
        y[i] += x[i]
}

int blockSize = 256;
int numSMs;
cudaDeviceGetAttribute(&numSMs,
    cudaDevAttrMultiProcessorCount, devId);
add<<<32*numSMs, block_size>>>(n, d_x, d_y);
```


- This is a *grid stride* loop (grid dim = number of blocks launched by

Threads, blocks, grids

CUDA heirarchy

- Threads (arranged in warps) belong to blocks
- Blocks (assigned to SMs) belong to grids
- Layout of block, grid, thread IDs can be up to 3D

Threads, blocks, grids

```
// Consider time-stepping a 96-by-96 grid equation
dim3 dimGrid(6,6,1);
dim3 dimBlock(16,16,1);
wave2Dkernel<<<dimGrid,dimBlock>>>(...);
```

- Thread indexing is “row major”
  - Consecutive threads have successive values of `threadIdx.x`
  - Mostly matters for memory coalescence
- Allow `gridDim.x` in 1 to $2^{31} - 1$, others in 1 to $2^{16} - 1$.
- Total size of any block is limited to 1024 threads

How big?

- Many threads is fine if they don’t use too much fast memory

Wave

1D Wave

Continuous:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$$

with $u(0) = u(1) = 0$.

(Think 2D-3D — 1D is to make it all fit on slides.)

1D Wave

Discrete:

$$(\Delta t)^{-2}\left(u_{t+1,k} - 2u_{t,k} + u_{t-1,k}\right) = c^2(\Delta x)^{-2}\left(u_{t,k+1} - 2u_{t,k} + u_{t,k+1}\right)$$

with $u_{t,0} = u_{t,N} = 0$.

Becomes:

$$u_{t+1,k} = 2(1 - C^2)u_{t,k} - u_{t-1,k} + C^2(u_{t,k+1} + u_{t,k-1})$$

where $C = c(\Delta t/\Delta x)$. Let $D = 2(1 - C^2)$.

First code

```
void step(int nx, float C2, float D,
          float* restrict unext, // u_{t+1,:}
          float* restrict u,     // u_{t,:}
          float* restrict uprev) // u_{t-1,:}
{
    for (int k = 1; k < nx-1; ++k)
        unext[k] = D*u[k] - uprev[k] + C2*(u[k+1] + u[k-1]];
}
```

- Implements exactly the time step from math
- Assume `u[0] = u[nx] = 0` for BCs

Naive parallelism

```
void step(int nx, float C2, float D,
          float* restrict unext, // u_{t+1,:}
          float* restrict u,     // u_{t,:}
          float* restrict uprev) // u_{t-1,:}
{
    #pragma omp for
    for (int k = 1; k < nx-1; ++k)
        unext[k] = D*u[k] - uprev[k] + C2*(u[k+1] + u[k-1]];
}
```

Is this a good idea?

Many steps

```
void steps(int nx, int nt, float C2, float D, float* uu)
{
    for (int t = 0; t < nt; ++t) {
        float* unext = uu + ((t+1)%3) * nx;
        float* u     = uu + ( t   %3) * nx;
        float* uprev = uu + ((t+2)%3) * nx;
        for (int k = 1; k < nx-1; ++k)
            unext[k] = D*u[k] - uprev[k] + C2*(u[k+1] + u[k-1
    }
}
```

- Assume `uu` is length `3*nx` (pay attention to first two)
- Take `nt` time steps
- Do not need storage for all `nt`!

Tiling

- Each tile “owns” set of points to be updated
- Needs a “halo” of neighbors to compute (equal to `nt`)
- Tile i starts at `1+(i*(nx-1))/ntiles`
- Halo goes out as far as `nt` from edge

Tiling

To the board!

Tiling

```
void steps_tiled(int ntiles, int nx, int nt,
                 float C2, float D, float* u)
{
    float ulocal[3*TILEMAX];
    for (int i = 0; i < nt; ++i) {
        // Compute start/end of block owned and with halo
        // Copy data with halo into ulocal
        // Take nt steps
        // Copy data without halo back to u
    }
}
```

Assuming `steps_tiled` called in a parallel region:

```
void steps_tiled(int ntiles, int nx, int nt,
                 float C2, float D, float* u)
{
    float ulocal[3*TILEMAX];
    #pragma omp for
    for (int i = 0; i < nt; ++i) {
        // Compute start/end of block owned and with halo
        // Copy data with halo into ulocal
        // Take nt steps
        // Copy data without halo back to u
    }
}
```

CUDA wave

Basic strategy

- Want tiled time stepper
- Tiles are blocks
- Global solution in shared memory
- Local solutions per thread

Many steps (in tile)

```
__device__
void steps(int nx, int nt, float C2, float D, float* uu)
{
    for (int t = 0; t < nt; ++t) {
        float* unext = uu + ((t+1)%3) * nx;
        float* u     = uu + ( t   %3) * nx;
        float* uprev = uu + ((t+2)%3) * nx;
        k = threadIdx.x+1;
        if (k < nx-1)
            unext[k] = D*u[k] - uprev[k] + C2*(u[k+1] + u[k-1
    }
}
```

What goes wrong?

Many steps (in tile)

```
__device__
void steps(int nx, int nt, float C2, float D, float* uu)
{
    for (int t = 0; t < nt; ++t) {
        float* unext = uu + ((t+1)%3) * nx;
        float* u     = uu + ( t   %3) * nx;
        float* uprev = uu + ((t+2)%3) * nx;
        k = threadIdx.x+1;
        if (k < nx-1)
            unext[k] = D*u[k] - uprev[k] + C2*(u[k+1] + u[k-1
        __syncthreads();
    }
}
```

Across tiles

```
__global__
void steps_tiled(int ntiles, int nx, int nt,
                 float C2, float D, float* u)
{
    __shared__ float ulocal[3*TILEMAX];
    for (int i = 0; i < nt; ++i) {
        // Compute start/end of block owned and with halo
        // Copy data with halo into ulocal
        // Take nt steps
        // Copy data without halo back to u
    }
}
```

Bigger barriers?

- Cannot synchronize across blocks
- *Except* via kernel launch/end
- Don’t need to copy state back and forth every time

Recommendation

- NVidia resources are pretty good
- More pointers on NERSC docs page, too
- *Programming Massively Parallel Processors* (4e)

Have a great fall break!