Circuit programming models for GPUs

...

Vladimir Maksimovski, with much help from professor Pai

Why are GPUs useful

- I'll focus on NVIDIA Geforce 1080. Similar numbers for all NVIDIA GPUs.
- GPUs are specialized for graphics processing.
- This also makes them very good at SIMT (single instruction, multiple thread) operations.

What's inside a GPU (that SIMT operations care about)

- Video Ram (8 GB for Geforce 1080).
- Bus interface (320 GB/s for Geforce 1080).
- Many SMs (streaming multiprocessors) (20 for Geforce 1080).
- 128 CUDA cores per SM
  - marketing speak for an ALU (FP/INT).
- 96KB (on-chip!!) shared memory per SM.

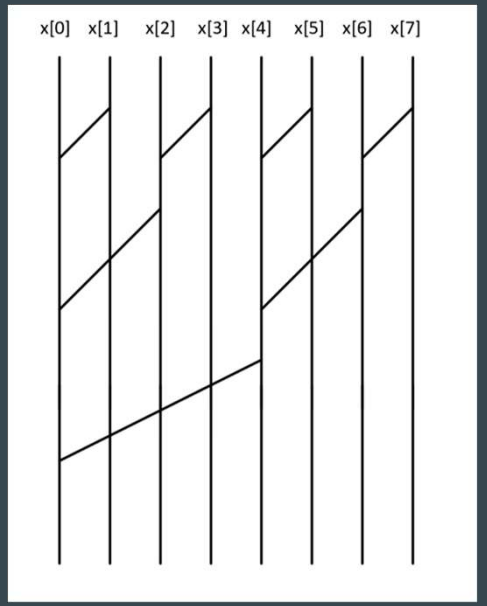
CUDA coding abstractions

- CUDA makes C++ to GPU easy.
- Programmer creates a kernel.
- Specifies #thread blocks, #threads.
- Thread blocks are independent, no shared memory.
- Threads can get shared memory, can synchronize.
- SM takes a thread block, executes it.
- SM splits this into 32-wide warps.
- Warps are executed concurrently.
- Memory allocation using cudaMalloc().

```
__global__ void add(int n, float *x, float *y, float *ans){
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if(i < n){
        ans[i] = x[i] + y[i];
    }
}

// x, y are pointers to vectors of size n.
float* vector_addition_using_gpu(int n, float *x, float *y){
    float *ans = (float *) malloc(n * sizeof(float));
    assert(ans != NULL);

    float *d_x, *d_y, *d_ans;
    cudaMalloc(&d_x, n * sizeof(float)); assert(d_x != NULL);
    cudaMalloc(&d_y, n * sizeof(float)); assert(d_y != NULL);
    cudaMalloc(&d_ans, n * sizeof(float)); assert(d_ans != NULL);

    cudaMemcpy(d_x, x, n * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, n * sizeof(float), cudaMemcpyHostToDevice);

    int grid_size = (n + 256 - 1) / 256;
    add<<<grid_size, 256>>>(n, d_x, d_y, d_ans);

    cudaMemcpy(ans, d_ans, n * sizeof(float), cudaMemcpyDeviceToHost);

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_ans);

    return ans;
}
```

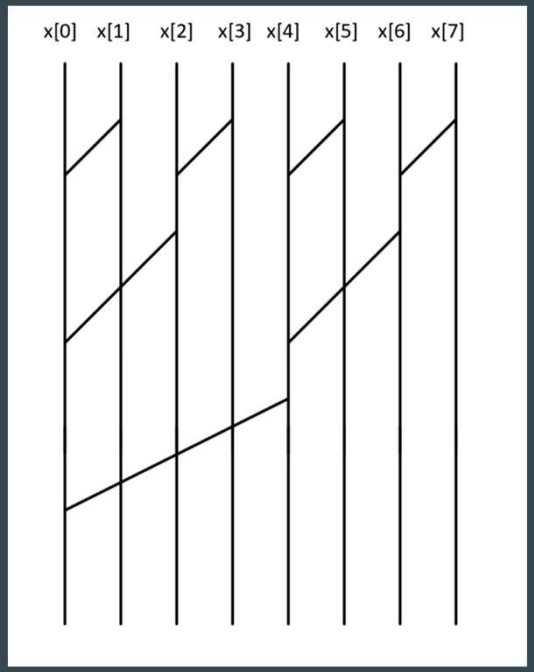
Okay, so what's up?

- C, C++, Fortran are designed for CPUs, not GPUs.
- Expressing GPU code in C++ hides a lot of the structure in favor of bit tricks.
- Consider computational circuits (reduce shown here).
- Input connects to output via circuit lines.
- Operation is a function of multiple lines.
- Operation written back to a single line.
- Using induction rules, circuits can be expressed easily.



Current work

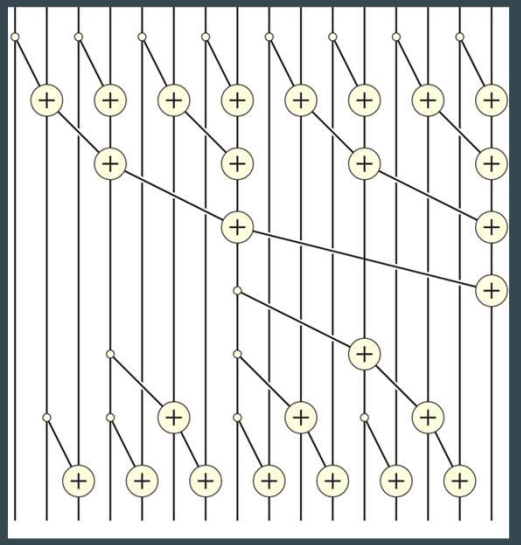
- Created a proof-of-concept circuit-to-cuda compiler using custom format.
- Limitations:
  - Assumes fixed circuit size.
  - Barriers after each step.
  - 2*n memory usage.




```
__global__ void sum(int *x0, int *x1){
    int tid = threadIdx.x;
    x1[tid] = x0[tid];
    switch(tid){
        case 0:
            x1[0] += x0[1];
            break;
        case 2:
            x1[2] += x0[3];
            break;
        case 4:
            x1[4] += x0[5];
            break;
        case 6:
            x1[6] += x0[7];
            break;
    }
    __syncthreads();
    x0[tid] = x1[tid];
    switch(tid){
        case 0:
            x0[0] += x1[2];
            break;
        case 4:
            x0[4] += x1[6];
            break;
    }
    __syncthreads();
    x1[tid] = x0[tid];
    switch(tid){
        case 0:
            x1[0] += x0[4];
            break;
    }
    __syncthreads();
    x0[tid] = x1[tid];
    switch(tid){
    }
}
```

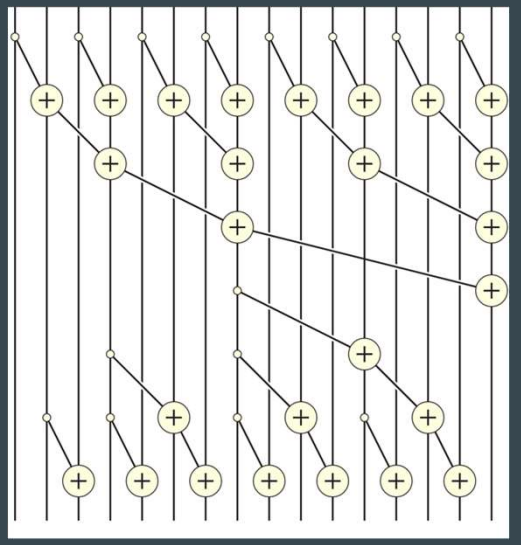
Optimizations feasibility pt.1

- Prefix sum takes in an array, and returns the prefix sum for all elements.
- Wrote this in CUDA manually. 34.40 GBPS
- Each thread block computes its prefix scan using ->
- Previous thread blocks aren't included in the sum!
- Prefix scan on thread block sums
- Propagate the thread-block-sums to each thread.



Optimizations feasibility pt.2

- Prefix sum takes in an array, and returns the prefix sum for all elements.
- Wrote this in CUDA manually. 34.40 GBPS
- Each thread block computes its prefix scan using ->
- Previous thread blocks aren't included in the sum!
- Prefix scan on thread block sums
- Propagate the thread-block-sums to each thread.
- Reduce thread divergence. 42.20 GBPS
- Reduce shared memory contention. 50.23 GBPS



Future work

- Consider what an inductive format would look like.
- See how to algorithmically express these optimizations

Thanks for listening!

SELECT questions

FROM audience