Graphics Report: GPU Accelerated Computing Using CUDA

Harshadeep Kambhampati (hk24873)
Kunal Tiwari (kt27627)
Alex Xu (ax748)
Ian Wang (ipw99)

I. Introduction

For our project, we worked on **designing and developing GPU-accelerated algorithms** for several different tasks by utilizing CUDA programming. Additionally, we wanted to benchmark our results across our CPU/GPU algorithm implementations to evaluate performance improvements (although definitely not an apples to applies comparison at all).

At a high-level, our goals for this project were to (1) learn CUDA programming concepts and syntax, (2) identify specific problems in which we could exploit parallelism using CUDA, and (3) actually learn how to break down linearly-designed CPU algorithms into parallelized GPU algorithms.

To that end, the four primary GPU-accelerated algorithms we wanted to look into were as follows:

1) Parallel Exclusive Prefix Sum (Scan)
2) Parallel Radix Sort
3) GPU-Accelerated N-Body Simulation
4) Convolutions Across Arrays

While the main focus of this project was on GPU-acceleration and designing CUDA-optimized algorithms, at a baseline this also required us to learn a basic implementation of each of these algorithms on the CPU in the first place. While an ideal comparison would require us to design optimal algorithms on both the CPU and GPU, in the interest of time and simply not having the knowledge needed, we went with more naive implementations of each of these algorithms (ex. did not use Barnes-Hut for n-body or implement both Blelloch and Hillis-Steele's for prefix sum).

The rest of this report will be organized as follows: Section II will cover some of the background context related to CUDA programming and parallelization broadly. Sections III to VI will respectively detail each of the GPU-accelerated algorithms mentioned and present our benchmarks. Lastly, section VII will summarize the key insights we have taken away from this project and discuss limitations of our work and benchmarks.

II. Relevant Background

A. What is CUDA?

CPUs tend to be composed of fewer, very powerful and versatile cores and hardware layouts great for general-purpose tasks with very low latency. However, GPUs are composed of far smaller, more specialized cores capable of achieving much greater throughput for tasks that can be parallelized. This includes areas like scientific computing, machine learning, graphics/image processing, and much more.

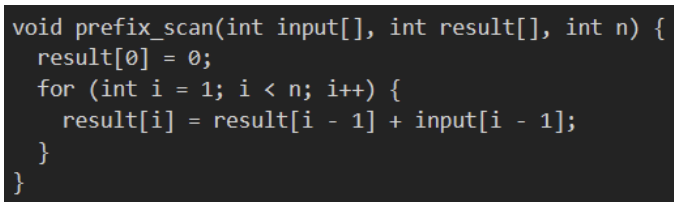
CUDA (which turns out doesn't stand for any particular acronym) is the platform designed by NVIDIA for programmers to easily interface between the CPU and GPU to leverage GPU parallelization and acceleration. Essentially, it allows you to write code on the CPU that can interface with the GPU to parallelize tasks. In the process of learning CUDA, there were a ton of important concepts and terminology we had to also understand.

B. Key CUDA Concepts

To give a VERY high-level summary, CUDA distinguishes code to run between the host (CPU) and device (GPU), allowing you to launch "kernel functions" that run as parallel threads on the GPU. The GPU is composed of thousands of smaller compute-specialized "CUDA cores" that are able to run these threads in parallel, thus making parallelizable tasks much faster. Units of execution on the GPU are organized as THREADS $<$ WARPS $<$ BLOCKS $<$ GRIDS. Warps are the smallest unit of execution on a single CUDA core at once (generally 32 threads), and blocks/grids tend to be more logical organizations for our purposes as CUDA programmers.

There are also ideas of "host, global, constant, and shared memory" in this CPU/GPU interfacing model of CUDA. The host memory refers to memory on the CPU's DRAM. The global memory refers to memory stored and allocated on the GPU's DRAM. Constant memory is just some read-only memory on the GPU that can also be accessed globally, but offers faster access time and can be broadcasted across WARPS efficiently. Shared memory is actually on-chip cache memory on each "streaming multiprocessor" (SM) of the GPU. The SMs assign workloads across different CUDA cores, and each CUDA core has access to shared memory on its respective SM, so logical blocks are able to use shared memory despite potentially executing on different cores.

Overall, there are tons of other details related to CUDA, but at a high-level these are some of the important ideas we used to optimally parallelize our algorithms on the GPU. Parallelization just refers to breaking down a linearly-executed task into smaller subtasks, or "kernel functions", that can run independently and simultaneously on the GPU.



```
void prefix_scan(int input[], int result[], int n) {
  result[0] = 0;
  for (int i = 1; i < n; i++) {
    result[i] = result[i - 1] + input[i - 1];
  }
}
```

Fig. 1. CPU Prefix Sum Implementation

III. Methodology

For all benchmarks conducted in this project, we ran our programs on the UTCS lab machines, each containing NVIDIA Quadro P2000 GPUs and Xeon E3-1270 v6 CPUs. We used the provided CUDA development 11.3.8 conda environment and timed all of our performance benchmarks using “chrono” from the C++ standard library to measure timing data. We ran our benchmarks 5 times and used the average of our results, testing across several different input sizes ”n” to observe how our algorithms scaled.

IV. Algorithm 1. Prefix Sum (Scan)

A. Overview

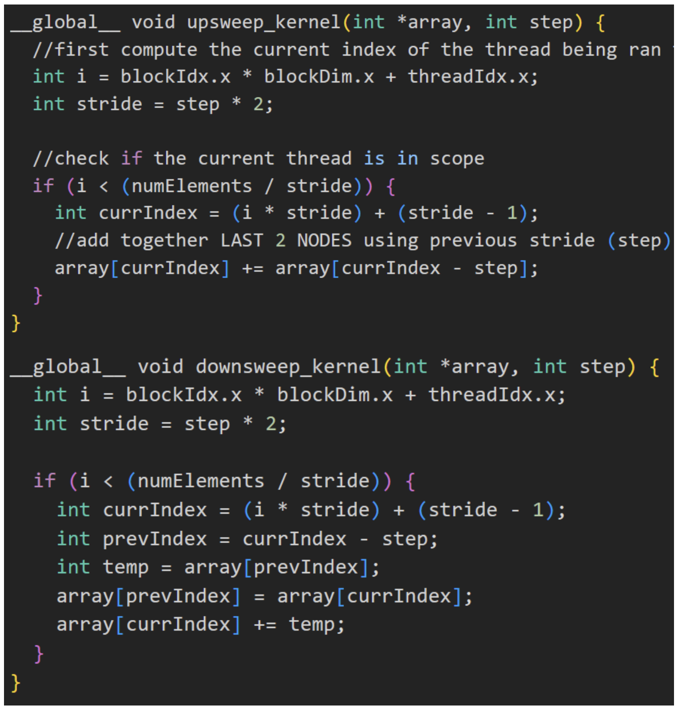
The prefix sum (scan) algorithm is the first of the algorithms we explored to design a GPU-accelerated solution around. To begin, the baseline algorithm essentially takes in some input array, and computes the cumulative exclusive sum of each element in the array. In other words, each index of the output array has the sum of all the prior elements from the input array (excluding the current element).

The CPU-based implementation of this algorithm is straightforward and linear in nature. It is an O(N) solution in which, starting from the first index until the last, simply add the previous element to the current element.

However, the classic prefix sum problem lends itself to be computed much faster when properly parallelized. In order to do this, we need to resolve this ”dependent” structure of the basic algorithm where each subsequent computation relies on the previous computation; we can’t execute such a task all in parallel if these dependencies haven’t yet been resolved.

For our CUDA-optimized implementation, we decided to use the Blelloch parallel prefix sum algorithm. Essentially, it architects a creative solution to parallelizing the prefix sum problem by breaking down this dependent chain of additions into two phases. The ”upsweep phase” essentially sums together elements at a particular ”step” value to build up a binary tree of partial prefix sums across steps, so to speak. This upsweep phase can occur in parallel across all indices at each step. Afterwards, the ”downsweep” phase propagates down these sums across the entire array to generate our final prefix sum array, which also occurs in parallel at each ”step”. The downsweep phase essentially sets the root of the binary tree to 0, then passes down each node’s value to the left child and the sum of the node and the left child to the right child.

While it’s difficult to explain and visualize, Fig 3 showcases an example of Blelloch’s algorithm being applied using both phases. The resulting array after both upsweep and downsweep phases is the correct prefix sum array. The main takeaway here is that we now have a way to at least parallelize each step of the prefix sum algorithm, thus resolving some of the dependencies to improve performance.



```
__global__ void upsweep_kernel(int *array, int step) {
  //first compute the current index of the thread being ran
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = step * 2;

  //check if the current thread is in scope
  if (i < (numElements / stride)) {
    int currIndex = (i * stride) + (stride - 1);
    //add together LAST 2 NODES using previous stride (step)
    array[currIndex] += array[currIndex - step];
  }
}

__global__ void downsweep_kernel(int *array, int step) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = step * 2;

  if (i < (numElements / stride)) {
    int currIndex = (i * stride) + (stride - 1);
    int prevIndex = currIndex - step;
    int temp = array[prevIndex];
    array[prevIndex] = array[currIndex];
    array[currIndex] += temp;
  }
}
```

Fig. 2. GPU Prefix Sum Upsweep and Downsweep for Each Step

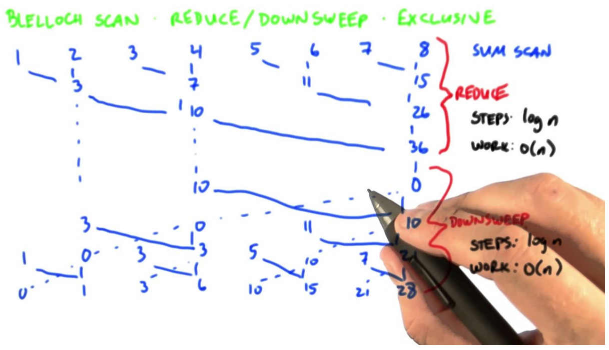


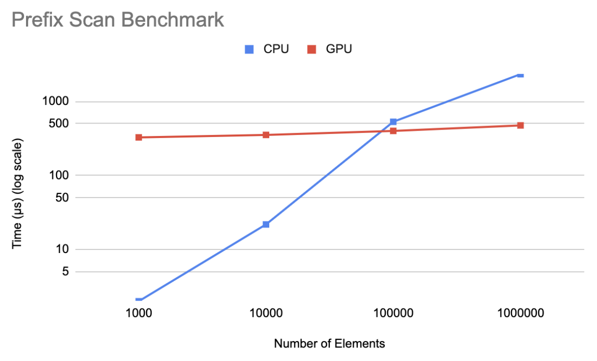
Fig. 3. Blelloch’s Parallel Prefix Sum Algorithm

B. Results

Shown in Fig. 1 and 2 are our specific implementations of both algorithms.

For smaller array sizes, our GPU implementation suffers in comparison to the CPU implementation, given the additional overhead of kernel launch and data-movement. However, we start seeing performance gains for arrays of $N = 100,000$ elements, where the GPU implementation provided a 1.32x speedup. At $N = 1,000,000$, the GPU implementation provided 4.9x speedup.

Results



Elements	CPU (μs)	GPU (μs)
1,000	2	326.48
10,000	21.8	352.72
100,000	530	399.72
1,000,000	2329.2	474.12

TABLE I
PREFIX SCAN BENCHMARK RESULTS (TIME IN MICROSECONDS).

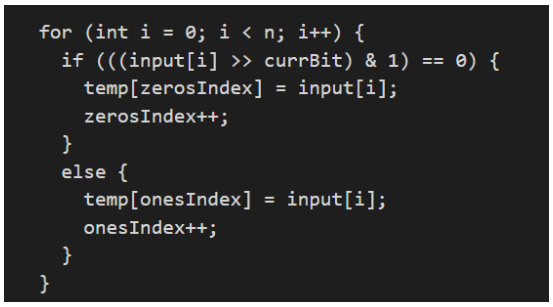
Overall, we saw vast performance improvement for our GPU-accelerated implementation of the prefix sum algorithm, and it served as an easy introductory algorithm for us to implement in CUDA without too much difficulty. Future work for this algorithm could include implementing the Hillis Steele algorithm for parallelized prefix sum as well to see if it offered even better performance benchmarks than Blelloch's.

V. ALGORITHM 2. RADIX SORT

A. Overview

The radix sort algorithm works by examining one bit at a time across all the numbers, grouping them by whether that bit is a 0 or a 1, and then reassembling the list. Because each pass over the data only looks at a single bit position, radix sort runs in $O(N \cdot B)$ time (where N is the number of elements and B is the number of bits, here 32).

1) **Bit–by–bit loop:** For each bit from 0 to 31, you scan the entire array to count how many numbers have a 0 in that bit position (call this count Z).
2) **Stable partition:** As pictured below, you make a temporary array and place all the "0"–bit numbers at the front (indices $0 \ldots Z-1$) and all the "1"–bit numbers at the back (indices $Z \ldots N-1$), preserving their original order.



```
for (int i = 0; i < n; i++) {
  if (((input[i] >> currBit) & 1) == 0) {
    temp[zerosIndex] = input[i];
    zerosIndex++;
  }
  else {
    temp[onesIndex] = input[i];
    onesIndex++;
  }
}
```

3) **Copy-back:** You then copy that temporary array back into the original array.

All of these steps are pure loops over N elements, done sequentially on one core. You pay for three full passes over the array per bit (count zeros, partition, copy back), so roughly 96 passes total. That's simple to understand, but it can be slow when N is large.

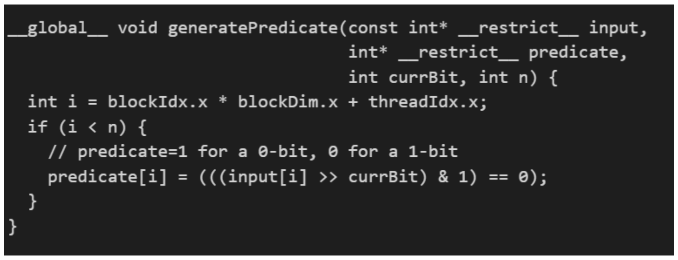
GPUs shine when the same operation needs to be done independently on lots of data. Here, for each bit we:

- Check each element's bit and record a 0/1 predicate.
- Do a parallel prefix–sum (scan) on that predicate array to figure out each element's destination index.
- Write (scatter) each element into its new position.

None of these steps cares about anything other than each element's own bit or the aggregated counts. That means we can assign one GPU thread to each array element and run all N bit–checks at the same time, instead of one after the other.

CUDA-optimized implementation

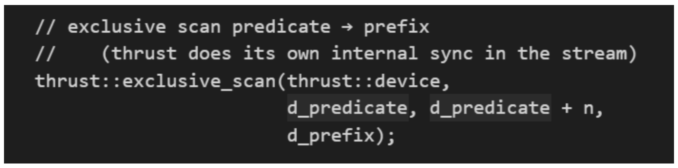
1) **Predicate generation:** Each thread reads one input value, extracts the current bit, and writes a 1 if it's 0 or 0 if it's 1.



```
__global__ void generatePredicate(const int* __restrict__ input,
                                  int* __restrict__ predicate,
                                  int currBit, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    // predicate=1 for a 0-bit, 0 for a 1-bit
    predicate[i] = (((input[i] >> currBit) & 1) == 0);
  }
}
```

Fig. 4. Generating a Predicate

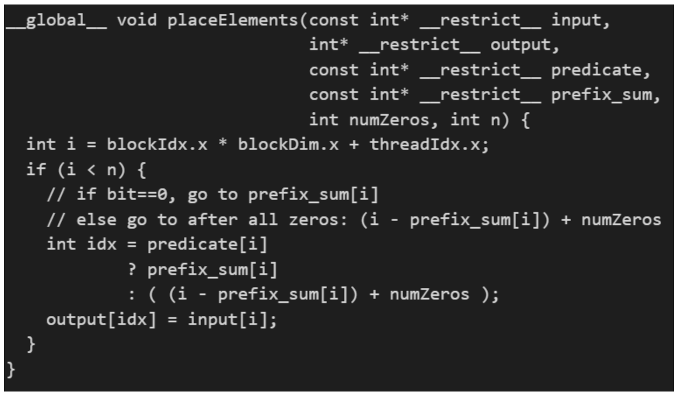
2) **Parallel scan:** Thrust's built-in scan does a tree–based algorithm, giving every thread its prefix sum in $O(\log N)$ steps instead of $O(N)$.



```
  // exclusive scan predicate → prefix
  //    (thrust does its own internal sync in the stream)
  thrust::exclusive_scan(thrust::device,
                         d_predicate, d_predicate + n,
                         d_prefix);
```

Fig. 5. Using Thrust's Prefix Sum

3) **Scatter:** Each thread computes its output index based on the predicate and prefix sum, then writes the element there.

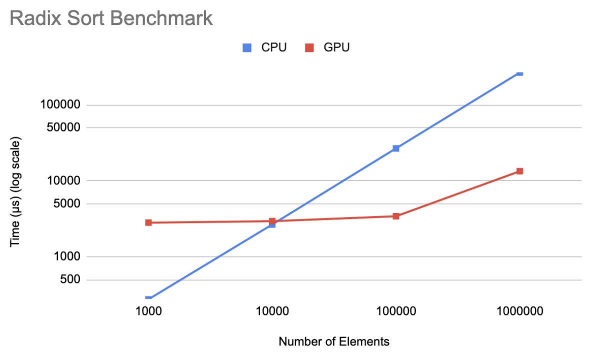


```
__global__ void placeElements(const int* __restrict__ input,
                              int* __restrict__ output,
                              const int* __restrict__ predicate,
                              const int* __restrict__ prefix_sum,
                              int numZeros, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) {
    // if bit==0, go to prefix_sum[i]
    // else go to after all zeros: (i - prefix_sum[i]) + numZeros
    int idx = predicate[i]
          ? prefix_sum[i]
          : ( (i - prefix_sum[i]) + numZeros );
    output[idx] = input[i];
  }
}
```

Fig. 6. Scattering Based on the 2 Previous Outputs

Between these kernels, we swap input and output pointers and repeat for all 32 bits. The work per bit is still $O(N)$, but now each of the $O(N)$ tasks is handled in parallel by thousands of GPU threads.

B. Results



Elements	CPU (μs)	GPU (μs)
1,000	275.0	2,818.0
10,000	2,658.8	2,948.8
100,000	26,749.6	3,421.2
1,000,000	266,837.4	13,411.0

TABLE II
RADIX SORT BENCHMARK RESULTS (TIME IN MICROSECONDS).

For small input sizes (e.g. $N = 10^3$), the GPU implementation is dominated by kernel launch and data-movement overhead, resulting in a slower runtime on the GPU (2.82 ms) compared to the CPU (0.275 ms). As the problem size grows, the GPU overhead is amortized:

$$\text{Speedup}(N) = \frac{T_{\text{CPU}}(N)}{T_{\text{GPU}}(N)}.$$

- For $N = 10^4$:

$$\text{Speedup} = \frac{2\,658.8}{2\,948.8} \approx 0.90,$$

so the GPU and CPU runtimes are comparable.
- For $N = 10^5$:

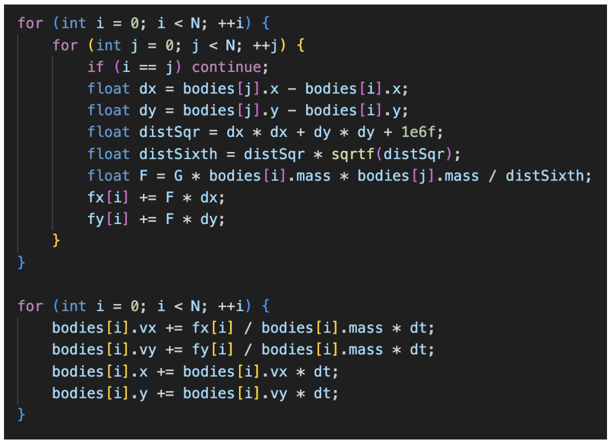
$$\text{Speedup} = \frac{26\,749.6}{3\,421.2} \approx 7.82,$$

indicating the GPU is nearly eight times faster.
- For $N = 10^6$:

$$\text{Speedup} = \frac{266\,837.4}{13\,411.0} \approx 19.90,$$

demonstrating almost a twenty-fold improvement.

These results confirm that while GPUs incur fixed overheads, they deliver substantial performance gains for large data sizes thanks to massive parallelism in the bit-wise scanning and scatter operations. The crossover point—where GPU performance surpasses the CPU—lies between 10^4 and 10^5 elements in this benchmark.



```
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        if (i == j) continue;
        float dx = bodies[j].x - bodies[i].x;
        float dy = bodies[j].y - bodies[i].y;
        float distSqr = dx * dx + dy * dy + 1e6f;
        float distSixth = distSqr * sqrtf(distSqr);
        float F = G * bodies[i].mass * bodies[j].mass / distSixth;
        fx[i] += F * dx;
        fy[i] += F * dy;
    }
}

for (int i = 0; i < N; ++i) {
    bodies[i].vx += fx[i] / bodies[i].mass * dt;
    bodies[i].vy += fy[i] / bodies[i].mass * dt;
    bodies[i].x += bodies[i].vx * dt;
    bodies[i].y += bodies[i].vy * dt;
}
```

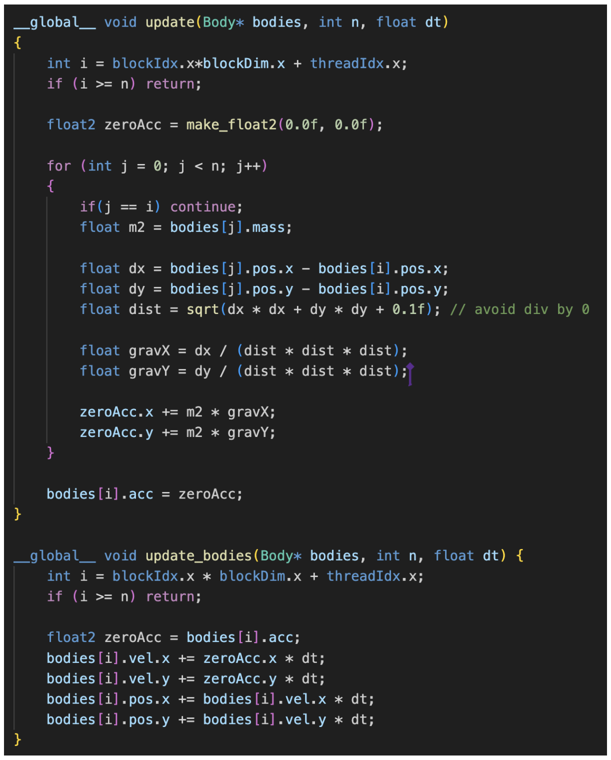
Fig. 7. CPU N-Body Implementation

VI. Algorithm 3. N-Body Simulation

The N-body simulation is a computational method of modeling the dynamic behaviors of particles in a system. N-body simulations can range from simple to extremely complex, even being used to study phenomena like the evolution of galaxies. The main challenge here lies in computing the forces each body exerts on every other body, which requires pairwise calculations.

For the baseline CPU implementation of the N-body simulation (Fig 7), all we have to do is iterate over all N bodies and then do another loop over all the other particles to calculate the gravitational forces between bodies. This is a very brute-force method, which leads to $O(N^2)$ running time, because each body must be compared to every other body. This becomes extremely inefficient, especially when we get to huge simulations, such as simulating galaxies, which calls for another method to be used.

Due to its computational intensity and highly parallel nature, the N-body problem greatly benefits from GPU acceleration. To efficiently parallelize the algorithm, we need to resolve the inherent data dependency problem. Each body's updated



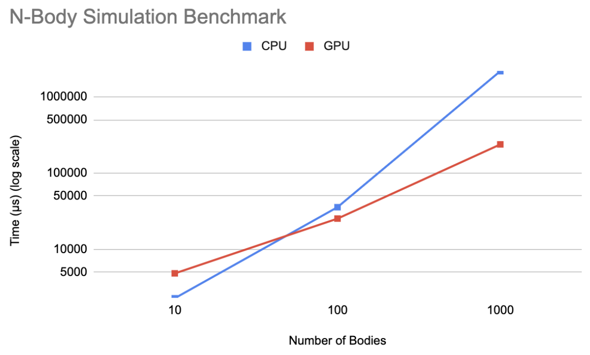
```
__global__ void update(Body* bodies, int n, float dt)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i >= n) return;

    float2 zeroAcc = make_float2(0.0f, 0.0f);

    for (int j = 0; j < n; j++)
    {
        if(j == i) continue;
        float m2 = bodies[j].mass;

        float dx = bodies[j].pos.x - bodies[i].pos.x;
        float dy = bodies[j].pos.y - bodies[i].pos.y;
        float dist = sqrt(dx * dx + dy * dy + 0.1f); // avoid div by 0

        float gravX = dx / (dist * dist * dist);
        float gravY = dy / (dist * dist * dist);

        zeroAcc.x += m2 * gravX;
        zeroAcc.y += m2 * gravY;
    }

    bodies[i].acc = zeroAcc;
}

__global__ void update_bodies(Body* bodies, int n, float dt) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i >= n) return;

    float2 zeroAcc = bodies[i].acc;
    bodies[i].vel.x += zeroAcc.x * dt;
    bodies[i].vel.y += zeroAcc.y * dt;
    bodies[i].pos.x += bodies[i].vel.x * dt;
    bodies[i].pos.y += bodies[i].vel.y * dt;
}
```

Fig. 8. GPU N-Body Implementation

position and velocity depend on calculations involving all other bodies, making it challenging to naively parallelize.

For our CUDA-optimized implementation (Fig 8), we adopted a strategy that leverages the parallel processing capabilities of GPUs. Instead of calculating interactions sequentially, each GPU thread independently computes the force on a single body from all other bodies simultaneously. Specifically, we partition the workload such that each CUDA thread corresponds to a particular body and independently computes the total gravitational acceleration on that body based on all other bodies. This parallelizes the most computationally intensive component, significantly improving execution performance.

A. Results



The advantage of parallelization is quite evident with the N-body simulation. At just $N = 1,000$ bodies for 100 steps, the GPU implementation already provided a 9x speedup, and the

Elements	CPU (μs)	GPU (μs)
10	2280.8	4842.4
100	35685.8	25326
1,000	2168263.6	237911.2

TABLE III
N-BODY SIMULATION BENCHMARK RESULTS (TIME IN MICROSECONDS).

CPU implementation actually timed out when we attempted $N = 10,000$ bodies. In addition, the GPU implementation was bottlenecked in that the calculated positions for each of the bodies still had to be written to a .json file, which is not parallelized.

B. Limitations

While we were able to implement the naive CPU implementation of the n-body algorithm, we weren't able to implement the optimal Barnet-Hut algorithm. As such, we did not have the most accurate representation of an ideal CPU baseline for the n-body simulation to compare to our GPU implementation. In the future, we would like to work on implementing and comparing using this more accurate baseline for the CPU algorithm.

VII. Algorithm 4. Convolutions

A. Overview

A convolution is just some way of multiplying some input array of data by some mask, or "kernel", centered about each element of the input array to produce an output array. In practice, convolutions are often used in machine learning applications and image processing tasks such as edge detection or sharpening/blurring. In the context of our project about GPU acceleration, convolutions tend to be extremely parallelalizable tasks because we are performing many different independent computations when computing the convolution centered around different elements.

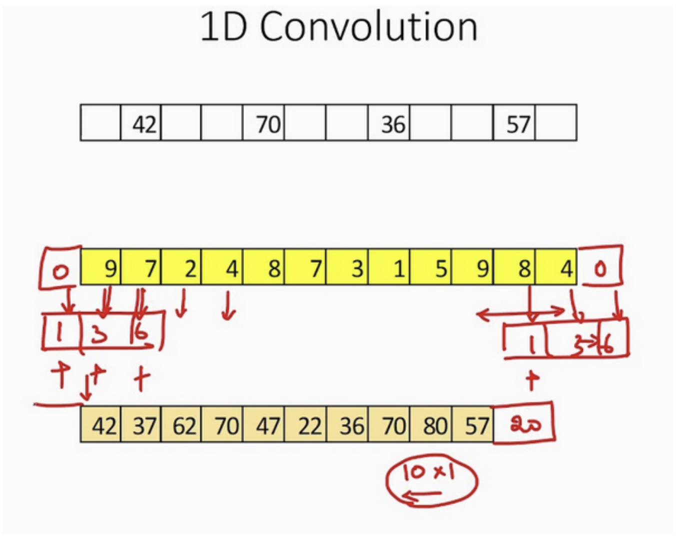
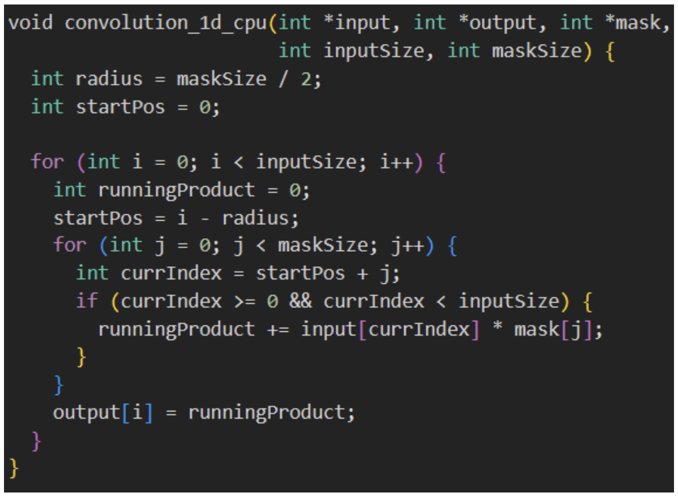


Fig. 9. 1D Convolution Example



```
void convolution_1d_cpu(int *input, int *output, int *mask,
                        int inputSize, int maskSize) {
  int radius = maskSize / 2;
  int startPos = 0;

  for (int i = 0; i < inputSize; i++) {
    int runningProduct = 0;
    startPos = i - radius;
    for (int j = 0; j < maskSize; j++) {
      int currIndex = startPos + j;
      if (currIndex >= 0 && currIndex < inputSize) {
        runningProduct += input[currIndex] * mask[j];
      }
    }
    output[i] = runningProduct;
  }
}
```

Fig. 10. CPU 1D Convolution Implementation

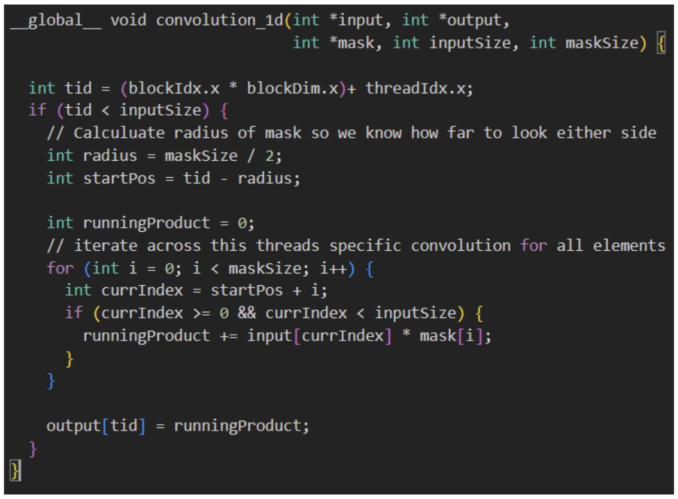
Shown in Fig 4 is a simple example of a 1D convolution given an input array and a mask of [1, 3, 6]. The mask is centered about each element in the input array, then aligned indices are multiplied together and summed up to produce an output for each centered index. Every single one of these computations is independent and thus serves as a good task for CUDA optimization on the GPU.

For our project, we focused on just implementing and optimizing this 1D convolution algorithm using input arrays of just random integers. Shown in Fig 5 is our implementation of the CPU 1D convolution algorithm. Given some input and mask arrays, we compute a ”runningProduct” between all elements (in valid range from 0 to inputSize) between the input and mask, shifting where we center the mask from index 0 to ”maskSize - 1”. This ends up being an O(NM) algorithm with respect to input and mask size.

For our CUDA-optimized solution, we started off with the simple parallelization of just computing the convolution centered about each element in the input array across parallel threads. As such, we wrote our kernel function such that the thread index determined which index of the input array to compute the convolution for, and wrote back each thread’s output to a global output array in parallel.a

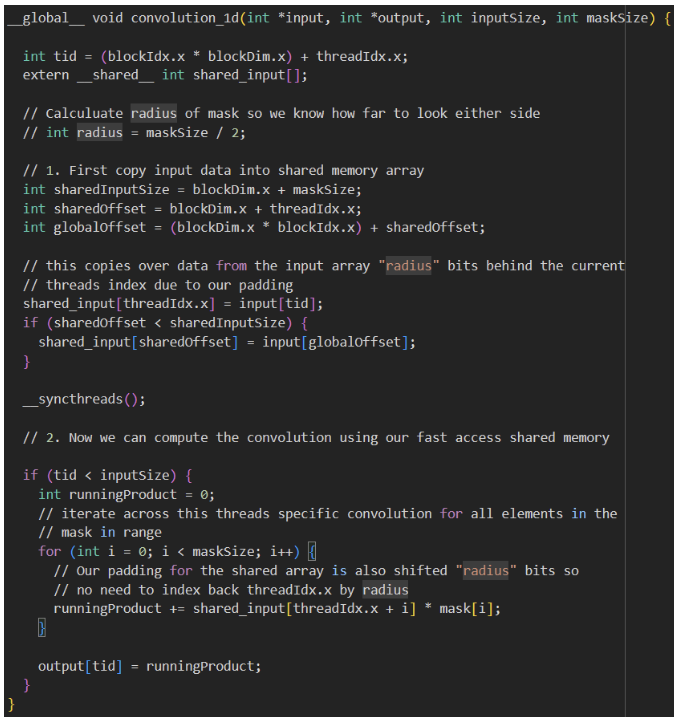
Shown in Fig 6 is our naive implementation of GPU-accelerated 1D convolution. We call this kernel function in a thread for every element of the input array. While this parallelization is already a good improvement, we also wanted to explore some of the additional techniques used in CUDA programming like shared/constant memory to see if we could apply them here. Currently, we allocate a copy of the input and mask arrays in global memory on the GPU, but global memory can be slow to access, especially when we’re accessing both arrays so many times across the entire convolution.

As such, we decided to further optimize our algorithm to better leverage GPU memory and hopefully demonstrate even further speedup. Shown in Fig 7 is our optimized implementation of GPU-accelerated 1D convolution.



```
__global__ void convolution_1d(int *input, int *output,
                               int *mask, int inputSize, int maskSize) {

  int tid = (blockIdx.x * blockDim.x)+ threadIdx.x;
  if (tid < inputSize) {
    // Calculuate radius of mask so we know how far to look either side
    int radius = maskSize / 2;
    int startPos = tid - radius;

    int runningProduct = 0;
    // iterate across this threads specific convolution for all elements
    for (int i = 0; i < maskSize; i++) {
      int currIndex = startPos + i;
      if (currIndex >= 0 && currIndex < inputSize) {
        runningProduct += input[currIndex] * mask[i];
      }
    }

    output[tid] = runningProduct;
  }
}
```

Fig. 11. GPU Naive 1D Convolution Implementation



```
__global__ void convolution_1d(int *input, int *output, int inputSize, int maskSize) {

  int tid = (blockIdx.x * blockDim.x) + threadIdx.x;
  extern __shared__ int shared_input[];

  // Calculuate radius of mask so we know how far to look either side
  // int radius = maskSize / 2;

  // 1. First copy input data into shared memory array
  int sharedInputSize = blockDim.x + maskSize;
  int sharedOffset = blockDim.x + threadIdx.x;
  int globalOffset = (blockDim.x * blockIdx.x) + sharedOffset;

  // this copies over data from the input array "radius" bits behind the current
  // threads index due to our padding
  shared_input[threadIdx.x] = input[tid];
  if (sharedOffset < sharedInputSize) {
    shared_input[sharedOffset] = input[globalOffset];
  }

  __syncthreads();

  // 2. Now we can compute the convolution using our fast access shared memory

  if (tid < inputSize) {
    int runningProduct = 0;
    // iterate across this threads specific convolution for all elements in the
    // mask in range
    for (int i = 0; i < maskSize; i++) {
      // Our padding for the shared array is also shifted "radius" bits so
      // no need to index back threadIdx.x by radius
      runningProduct += shared_input[threadIdx.x + i] * mask[i];
    }

    output[tid] = runningProduct;
  }
}
```

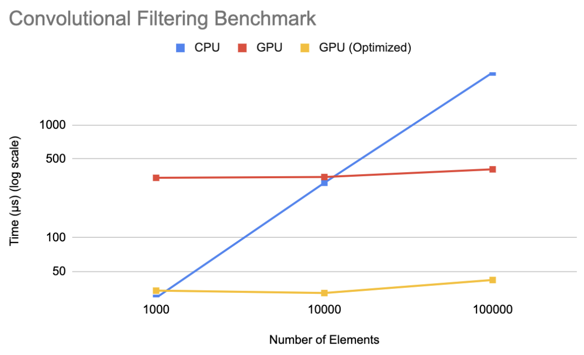
Fig. 12. GPU Memory-Optimized 1D Convolution Implementation

To begin, the first big change we made in this algorithm was allocating our mask array in constant memory. Since constant memory is faster global read-only memory on the GPU and the mask remains to be read-only data across the entire convolution, this was a simple optimization.

Additionally, we wanted to use shared memory across blocks of threads on the GPU for minimizing memory access latency. As such, we first needed to adjust our kernel function such that all of the threads first start off copying over data into their ”shared input” buffer so that the block has the entire range of the input array needed for its respective convolution. This copying process is also parallelized on the GPU. While it sounds easy, this copying process involved (1) adjusting our input data to include a mask buffer, (2) using some indexing tricks to copy over the entire range of data needed across our block with minimal ”warp divergence”, and then (3) adjusting how we index this data since it only represents part of our input array.

We make sure to syncthreads() after copying over all of our input data into our shared memory arrays to make sure all threads have finished copying over their data. Afterwards, the actual 1D convolution computation is almost the same as our naive implementation, just with some indexing adjustments.

B. Results



Elements	CPU (μs)	GPU (μs)	Optimzed-GPU (μs)
1000	29	340.2	33.6
10000	307.2	345.8	32
100000	2950.6	405	41.8

TABLE IV
CONVOLUTIONAL FILTERING BENCHMARK RESULTS (TIME IN MICROSECONDS).

Once again, performing convolutional filtering on smaller arrays was slower for the GPU, given the initial overhead. However, at $N = 10,000$ elements, our benchmarked times started evening out. At $N = 1,000,000$ elements, the GPU implementation yielded a 7.29x speed up. Furthermore, a refined GPU implementation which made use of shared memory achieved a 70.59x speed up.

C. Limitations

While we were able to implement and optimize 1D convolution implementations on the CPU/GPU, we ended up not having time to finish an optimal 2D convolution algorithm. We also wanted to demonstrate the use of 2D convolutions for image blurring and sharpening by passing in input images parsed into 2D arrays, but due to the limitations of libraries we had available to us on the UTCS lab machine conda environment, we weren't able to attempt this either.

Mainly, we just focused on the scope of our project being CUDA optimization, and as such most of the optimization ideas wouldn't have changed much with a 2D convolution algorithm. The indexing tricks between threads for parallelization would have just been harder to work out and implement.

VIII. DISCUSSION

Overall, this project has helped us learn a lot more about how CUDA programming and GPU parallelization works. We've also demonstrated some of the massive performance improvements that GPU-accelerated algorithms can demonstrate (at least across the four tasks we optimized for scaling at larger n values).

Beyond just learning about CUDA, as a secondary consequence of our goals, we also learned about the basic CPU implementations of each of the discussed algorithms as well (which honestly ended up taking just about as much effort to learn and implement properly).

A. Overall Limitations

While we had no issues with implementing and optimizing any of our CUDA code since we had access to the complete CUDA developer toolkit on the UTCS lab machines, it was unfortunate that we couldn't get access to any useful graphics libraries like OpenGL on the same conda environment. As such, we were very limited in terms of what kinds of visualizations we were able to generate with the algorithms we studied, which is a shame since the n-body simulation and convolution definitely could have produced some cool artifacts. Additionally, we could have seen the live-time difference in rendering output between the CPU and GPU implementations.

Additionally, a very important point to mention is that while our benchmarks were helpful in authenticating our own CUDA implementations and conveyed very general differences in magnitude, our benchmarking techniques were NOT at all perfect nor were they a fair apples-to-apples comparison. We used the very simple ”chrono” library for timing data, which includes kernel launch and thread synchronization overhead. Our implementations of the actual algorithms presented on the CPU/GPU also might not have been the most optimal implementations on either hardware, such as not using the Barnet-Hut algorithm for our CPU-based n-body simulation. As such, it's not an accurate reflection of the ideal performance of the CPU/GPU. In terms of actual hardware, we were benchmarking simply on whatever hardware was available on the UTCS lab machines, which may not have represented an equally-matched CPU/GPU comparison hardware wise. The point being, our benchmarks were mostly just an educational demonstration for our own understanding and do no perfectly represent the performance gap between CPU/GPU computing. However, we believe they still do convey a significant enough difference in magnitude such that the benefits of GPU-acceleration seem clear.

Lastly, we definitely had more future work we wanted to expand on for each of the algorithms presented, as discussed in their respective sections. Given more time, we'd love to further explore those future works.

All things considered, however, we're still very happy with how our project turned out and are excited to have had this exposure to CUDA programming broadly.

References

[1] "Making an N-Body Simulation." YouTube, YouTube, 2024, www.youtube.com/watch?v=L9N7ZbGSckkt=296sab$_channel$=Deadlock.

[2] "Cuda Crash Course." YouTube, YouTube, 2019, www.youtube.com/playlist?list=PLxNPSjHT5qvtYRVdNN1yDcdSl39uHV$_sU$.

[3] Mohan, Shivam. "Understanding Implementation of Work-Efficient Parallel Prefix Scan." Medium, Nerd For Tech, 4 Apr. 2023, medium.com/nerd-for-tech/understanding-implementation-of-work-efficient-parallel-prefix-scan-cca2d5335c9b.