Optimization of Nanopond.c for GPU Acceleration

Authors: Stuart Sessions, Walt Jones, Blanche Stora, Nathan Curl, Barry Rountree, Danielle Ellsworth

Affiliations: Colorado College Computer Science Department, Lawrence Livermore National Labratory

Background

While CPUs are used more widely, GPUs are gaining in popularity, and are much faster at doing some operations (mainly matrix multiplications). We attempted to optimize nanopond.c for execution on a GPU, while maintaining the same logic to produce identical results.

Abstract

We used a variety of optimization techniques and code changes to port a 90's piece of C simulation software to run on the GPU. This included removing if-statements from the code, changing memory allocation, and redesigning the execution pattern of the cells. In the limited time spent at Lawrence Livermore National Laboratory, we were able to successfully compile the program to run on a single thread of a GPU. We continue to work to debug and verify our results before we implement multithreading.

Methods & Results

1. Code Debranching

Modern CPUs use extra transistors and more complicated architecture to quickly and efficiently handle branching. To keep cost, and power down while maximizing performance, GPUs do not have this “smart” logic. The main execution of the nanopond is a large switch statment which identifies the genome’s instruction and runs it. To efficiently run nanopond on the GPU we determined we needed to remove this branching using logical operators instead of if, switch, and ternary statments. This is done by changing all variables for every time the genome executes on instruction. By ensuring unused variables stay the same, and setting used variables to the desired value, we succeeded in keeping the deterministic randomness of nanopond intact.

Before “de-branching the code, variables were set in cases:

```
switch(inst) {
    case 0x0: /* ZERO: Zero VM state registers */
        reg = 0;
        ptr_wordPtr = 0;
        ptr_shiftPtr = 0;
        facing = 0;
        break;
}
```

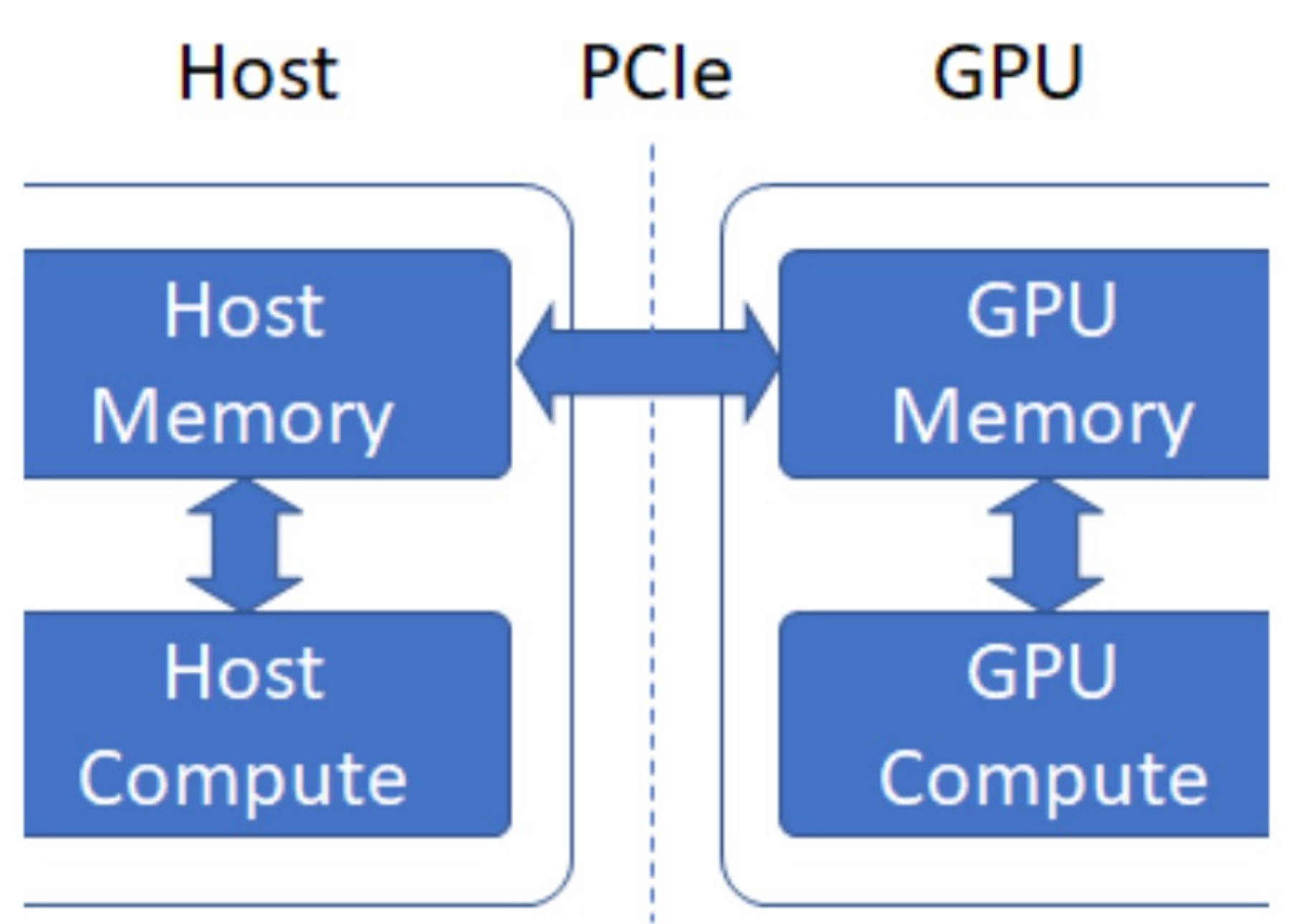
We use logical “or” operators to make sure the variables are only changed when “inst” is the correct value.

```
reg= (inst == 0x1 || inst == 0x2 || inst == 0x6 || inst == 0x8 || inst == 0x9 || inst == 0xa || inst ==
 0xb || inst == 0xd ||inst == 0xe || inst == 0xf) * (reg) +
((inst==0x0)*0) +
((inst==0x3)*((reg + 1) & 0xf)) +
((inst==0x4)*((reg - 1) & 0xf)) +
((inst==0x5)*((pptr->genome[ptr_wordPtr] >> ptr_shiftPtr) & 0xf)) +
((inst==0x7)*((outputBuf[ptr_wordPtr] >> ptr_shiftPtr) & 0xf)) +
((inst==0xc)*((pptr->genome[wordPtr] >> shiftPtr) & 0xf));
ptr_shiftPtr = (inst == 0x3 || inst == 0x4 || inst == 0x5 || inst == 0x6 || inst == 0x7 || inst ==
0x8 || inst == 0x9 || inst == 0xa || inst == 0xb || inst == 0xc || inst == 0xd || inst == 0xe || inst
== 0xf) * (ptr_shiftPtr) +
((inst == 0x0)*0)+
((inst == 0x1)*((ptr_shiftPtr+4)*((ptr_shiftPtr+4)<SYSWORD_BITS)))+
((inst == 0x2)*(((ptr_shiftPtr==0)*SYSWORD_BITS)+ptr_shiftPtr-4));
ptr_wordPtr = (inst == 0x3 || inst == 0x4 || inst == 0x5 || inst == 0x6 || inst == 0x7 || inst == 0x8
|| inst == 0x9 || inst == 0xa || inst == 0xb || inst == 0xc || inst == 0xd || inst == 0xe || inst ==
0xf) * (ptr_wordPtr) +
((inst == 0x0)*0) +
((inst==0x1)*(((ptr_wordPtr*(ptr_shiftPtr!=0||((ptr_wordPtr+1)<POND_DEPTH_SYSWORDS))+(ptr_shiftPtr==0)*
((ptr_wordPtr+1)<POND_DEPTH_SYSWORDS)))))+
((inst==0x2)*(((ptr_wordPtr==0&&ptr_shiftPtr==(SYSWORD_BITS-4))*(POND_DEPTH_SYSWORDS))+ptr_wordPtr-(ptr
_shiftPtr==(SYSWORD_BITS-4))));
```

2. GPU Memory Allocation

Memory addresses are the computer's internal representation of where data is stored in it's memory. A device (GPU) accessed by the host (CPU) operates on different memory addresses, requiring the user to allocate memory in the host program specifically for the device to access. This is done using the “cudaMalloc” function.

In order to use values set on the host, the user must also copy the contents of the host memory into the address that will be used by the device. This is done using "cudaMemcpy."



(Below) Example of Memory Allocation

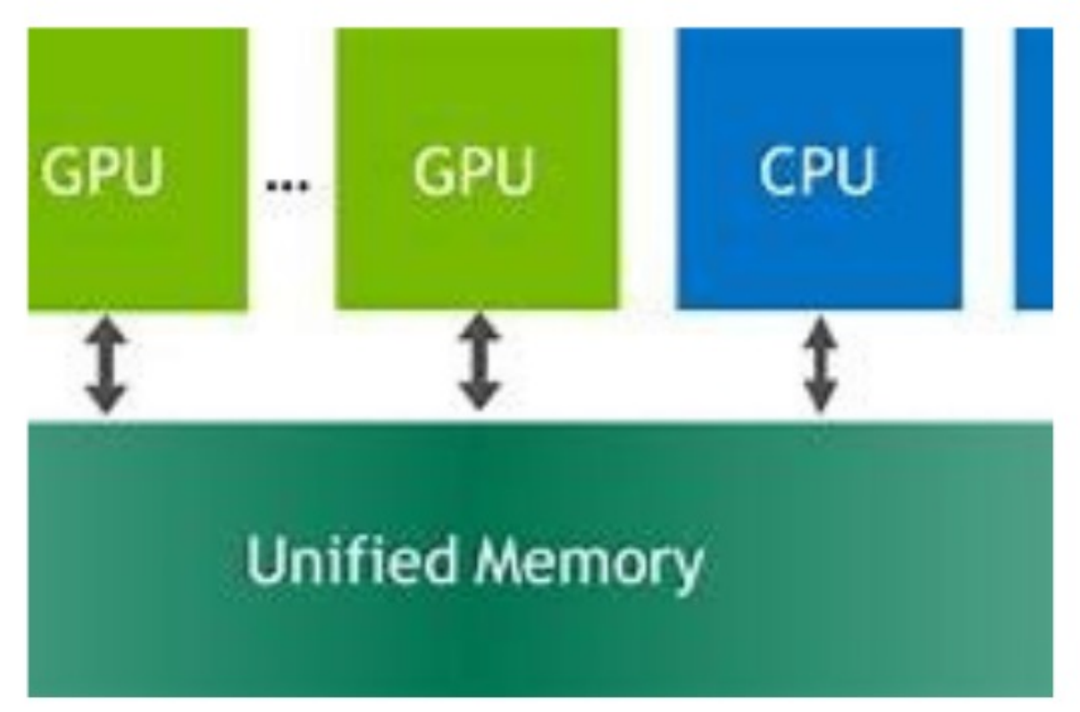
(Above) Visual representation of GPU vs host execution

```
struct Cell *d_pond; // initialize the device pond
cudaMalloc(&d_pond, POND_SIZE_X * POND_SIZE_Y * sizeof(struct Cell)); // allocate memory for the device
 pond on the GPU
initializePond<<<POND_SIZE_X, POND_SIZE_Y>>>(d_pond); // call the kernal to set cells to 0
struct Cell *h_pond = (struct Cell *)malloc(POND_SIZE_X * POND_SIZE_Y * sizeof(struct Cell)); //
initialize the host pond
run<<<1, 1>>>(d_pond, d_buffer, d_in, d_prngState, d_statCounters, cellIdCounter, d_accessAllowed1,
d_accessAllowed2); // call the run kernal
cudaMemcpy(h_pond, d_pond, POND_SIZE_X * POND_SIZE_Y * sizeof(struct Cell), cudaMemcpyDeviceToHost); //
 copy the changed pond back to the device
```

Another approach we took to making data accessible between a host and device is utilizing CUDA’s Unified Memory, which automatically allocates defined variables to be accessible to both a GPU and CPU.

```
/* This is used to generate unique cell IDs */
__managed__ uint64_t cellIdCounter = 0;
```

Defining a variable to be allocated through Unified Memory

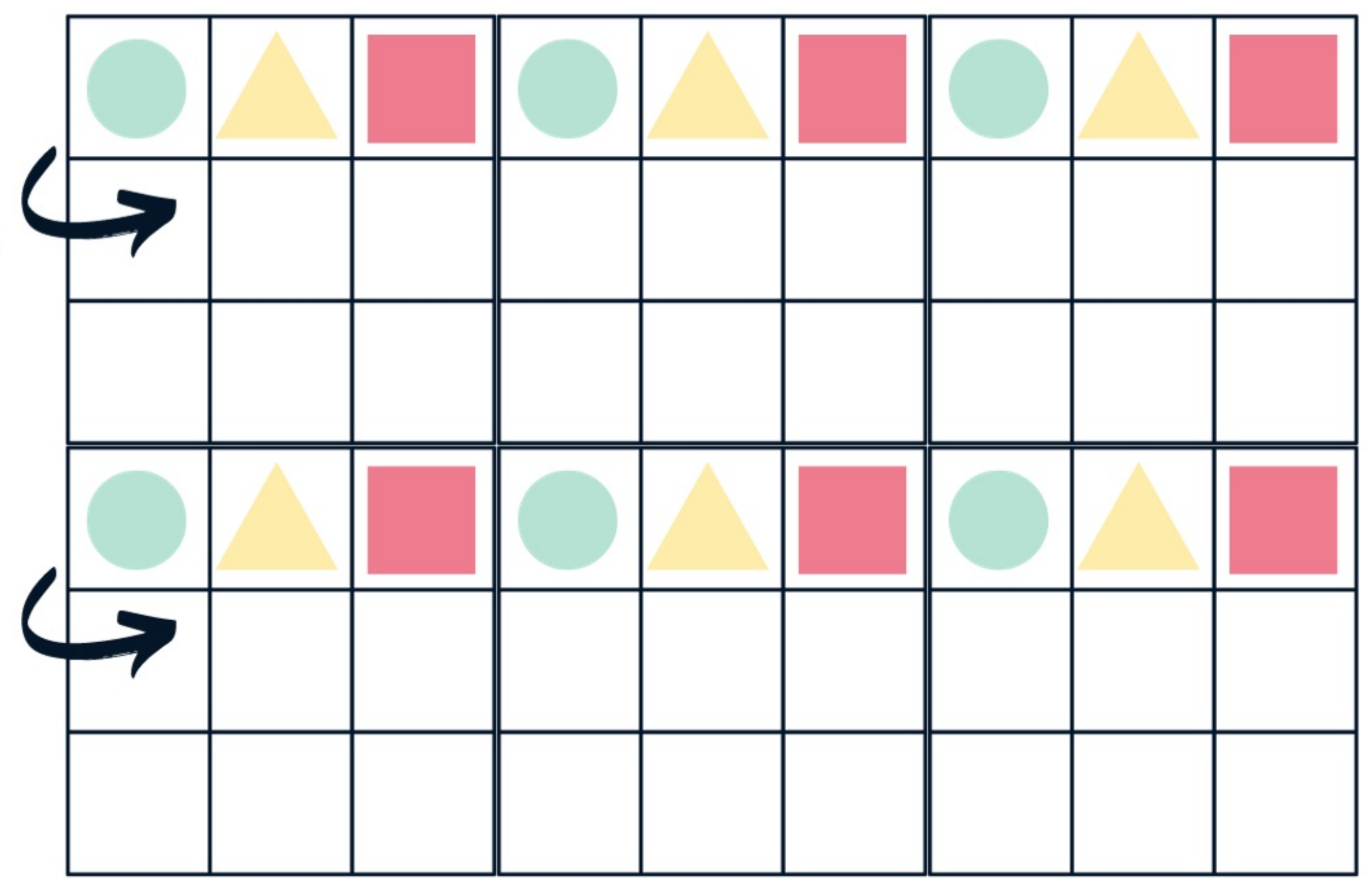


User interpretation of CUDA Unified Memory

Future Work

Once we achieve reliable single thread performance on a GPU, we will implement GPU multithreading to dramatically increase perforamance. The method we plan to use to achieve this while avoiding overwriting adjacent cells is a random stencil method. A cell may read or write on an adjacent cell, so we need to guarantee that no cells executing in the same warp will ever be within two cells of one another. Then, to avoid the “first execution” problem, we will randomly move the “stencil” around the board. This will guarantee that no cells overwrite eachother in the same execution cycle. However, we do recognize this will fundamentally change the nature of the nanopond simulation. In the original code, a cell was randomly picked to execute. In our proposed change, every cell will execute before a cell gets a second chance to execute. To measure whether the simulation archives the same results we will measure the genome distribution after running the original code and after our modifications. We propose that if the genome distributions, on a macro scale, are similar, the simulation’s outcomes remain similar.

Every color is one “warp” where every cell is executed by one thread at the same time. Then the next “warp” will excute, so on and so forth. We will randomly pick a “start cell” and apply this as the stencil to the board.



Conclusion

GPUs promise massive performance improvements by running static code through thousands of less powerful, specially designed cores. Our project barely touched the surface of GPU programming as we spent the majority of our time refactoring the code to a format easily run on this new architecture. Our code is nearly at a point where it could be easily run on this architecture. However, much of the challenge of programming GPUs is in the memory allocation and additional optimizations. That work will need to happen before nanopond runs efficiently on this hardware.

A huge thank you to Dr. Danielle Ellsworth, Dr. Barry Rountree, and the entire Livermore Computing Staff for hosting our visit and allowing us access to their systems.