THE UNIVERSITY OF CHICAGO

A PARTICLE IN CELL PERFORMANCE MODEL ON THE CS-2

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
BACHELOR OF ARTS IN COMPUTER SCIENCE WITH HONORS

DEPARTMENT OF COMPUTER SCIENCE

BY
MANDY LA

CHICAGO, ILLINOIS
JUNE 2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I could not have successfully completed this thesis without the help and support of the following people.

I would like to thank my advisor, Professor Andrew A. Chien, for his guidance through each stage of my research and writing process. Your enthusiasm for innovation in computer science inspires me to continue learning and working hard.

I would like to acknowledge my colleagues at Cerebras Systems for generously sharing information about the CS-1 and CS-2 with our research team. I would particularly like to thank Ilya Sharapov for the many productive discussions we have had regarding the ideas expressed in this thesis, Robert Schreiber for being our key advocate, and Michael James for lending us his technical expertise.

In addition, I would like to thank my friends and family for their constant support. You might not always understand what I am working on, but your presence is always comforting. Thank you, all.

ABSTRACT

As the once rapid improvement of general purpose computers slows due to the end of Moore's Law and Dennard Scaling, computer scientists must find novel methods to meet the world's hunger for computing power. The CS-2 by Cerebras Systems approaches this problem from a fresh perspective. The CS-2 packs 850,000 processing elements (PEs) onto one 462 sq cm Wafer Scale Engine. The combination of highly parallel computation, fast distributed memory, and efficient communication makes the CS-2 a fitting machine to run one of the most widely used algorithms in computational plasma physics, Particle In Cell simulations. We discuss two viable mappings of Particle In Cell onto the CS-2, the Static Particle Method and the Dynamic Particle Method. Using the Dynamic Particle Method, we present a theoretical peak performance model and an empirical performance model based on a partial implementation. The CS-2 achieves 3.67 PFLOPS and 1.72 PFLOPS in the theoretical and empirical performance models, respectively. We also consider additional complications such as load imbalance and particle movement between PEs. Accounting for these complications, the CS-2 achieves 1.66 PFLOPS in the theoretical performance model. This is three orders of magnitude greater performance than GPUs and CPUs on the market today. Our findings show the potential for the CS-2 in computing Particle In Cell codes, and we hope to inspire future implementations of Particle In Cell on the CS-2.

CHAPTER 1

INTRODUCTION

As general purpose computers have reached the end of the monumental age of rapid and consistent improvement driven by Moore's Law and Dennard Scaling, computer scientists must look to more specialized computing devices. These types of computers are called accelerators and allow computers to achieve higher and faster performance by exploiting the structure and operations of specific computations.

Graphics processing units (GPUs) are one example of an accelerator. GPUs are built with more arithmetic-logic units (ALUs) than central processing units (CPUs) in order to exploit parallelism and achieve higher operation rates. This is a desirable trait for computations like machine learning, dense and sparse linear algebra computations, spectral methods, and more.[1, 3] These are highly-relevant and impactful classes of computation in the modern-age of technology. For example, machine learning opens up the uses of computers from just computing to inferring. Given large amounts of data, which we have ample access to in the modern day, the computer is able to not only perform operations but use the results of those operations to inform future computations. The uses of this technology are unbounded and have been steadily creeping into our lives for over a decade. Some examples include self-driving cars, personalized ads online, virtual assistants, and facial recognition software.

Such a variety of potential uses drives companies to build bigger, better, and faster accelerators. Currently, the primary tactic for solving large parallel computations consists of clusters of GPUs wired together and attached to an external memory system. Performance on GPU clusters does not scale proportionally and much is lost in these cluster systems.[4] Thus computer engineers are driven to explore novel ideas that will allow their accelerators to tackle even the largest of problems.

One such approach, and the focus of this thesis, is wafer-scale processors. The word wafer here refers not to a sweet, thin cookie, but a wafer of thin silicon used for the fabrication

of integrated circuits. A wafer is around 300mm in diameter and typically makes many microcircuits that are later separated by wafer dicing and packaged as an integrated circuit. A typical GPU is at most around 900mm^2. A wafer-scale processor is one that utilizes the entire 70,650mm^2 wafer to make a single large chip. The motivation for this thesis is to demonstrate the potential of a wafer-scale processor to achieve impressive performance on impactful, real-world applications.

CHAPTER 2

THE CS-2 WAFER SCALE ENGINE

The CS-2, a system created by Cerebras Systems that packs powerful computing onto a single 462 sq cm Wafer Scale Engine (WSE), is a promising new approach to parallel accelerators. Instead of cutting up a silicon wafer into tiny microprocessors, Cerebras Systems packs an impressive 850,000 PEs onto one large chip. Figure 2.1 shows the CS-2 and the WSE it contains. The CS-2 houses the WSE-2 along with cooling mechanisms and power supply. All compute-hardware is located on the WSE-2.

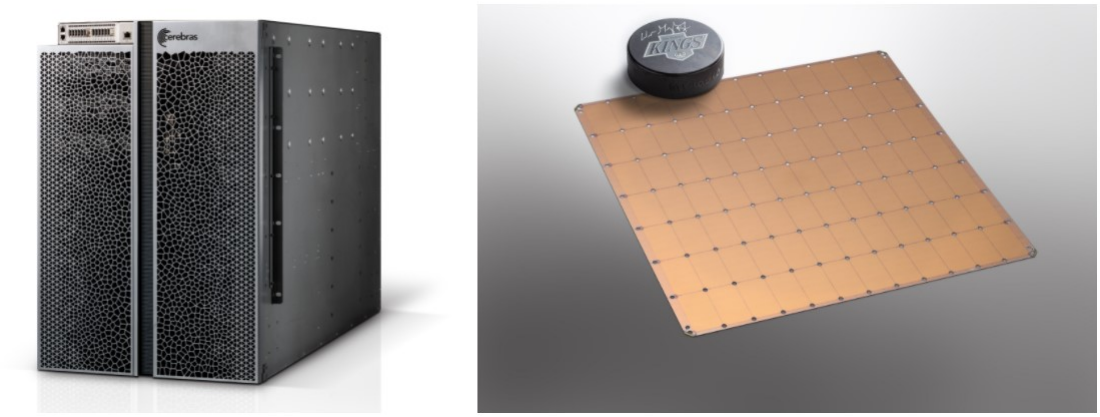

Figure 2.1: The CS-2 (left) houses the WSE-2 (right) along with cooling mechanisms and power supply. All compute-hardware is located on the WSE-2. The WSE-2 is shown next to a hockey puck for size comparison. The WSE-2 is 8.5 inches on both sides. The CS-2 is 15 Rack Units. Photos Courtesy of Cerebras Systems.

The strength of the WSE lies in its impressively low memory latency and network communication cost. The network fabric can communicate with all 850,000 PEs without ever leaving the chip. This mechanism gives the CS-2 a memory bandwidth of 20 Petabytes/second and a 27.5 Petabytes/second fabric bandwidth. These are the numbers marketed by Cerebras Systems and confirmed by independent benchmarking done by our research group. Table 2.1 gives an overview of the CS-2 compared to its first-generation version, the CS-1.

On this oversized chip is a highly parallel, distributed memory architecture. All memory within the CS-2 is on-chip, producing more memory bandwidth, single cycle memory latency,

	CS-1 (WSE-1)	CS-2 (WSE-2)	Difference
Number of Cores	400,000	850,000	2.1x
On-Chip Memory	18 GB	40 GB	2.2x
Memory Bandwidth	9 PB/s	20 PB/s	2.2x
Fabric Bandwidth	100 Pb/s	220 Pb/s	2.2x
Transistor Count	1.2 Trillion	2.6 Trillion	2.2x
Silicon Area	46,225 mm^2	46,225 mm^2	1.0x
Power Consumption	15 kW	17 kW	1.1x
IC Process	16 nm	7 nm	N/A

Table 2.1: The newly announced CS-2 more than doubles the potential of the first-generation CS-1. Although the physical size of the system stays the same, the massive increase in core count is thanks to the decrease in IC Process, allowing Cerebras Systems to fit 2.1 times more PEs on the WSE-2.

and lower energy cost for memory access. Each PE sits on what Cerebras calls a tile. Every tile contains the processor, memory, and a router. Figure 2.2 diagrams the structure of a tile. There is no shared memory that all PEs can access. There is no central PE that directs other PEs. Each of the 850,000 PEs work in parallel, managing its own memory, and communicating with one another through the 2D-mesh interconnection fabric. In addition, each PE can be individually programmed.

The router is bidirectionally linked to the processor and the routers of the four neighboring tiles. Each of the five links can be used in parallel on every cycle. The router has hardware queues for its connection to the core and for each of a set of virtual channels, avoiding deadlock. There are 24 virtual channels on each PE, and all or a subset of them can be configured during compile time. No runtime software is involved with communication. All arriving data is deposited by the hardware directly into memory or other desired location. Routing is configured during compilation and sets up any routes needed for communication

between potentially distant PEs.

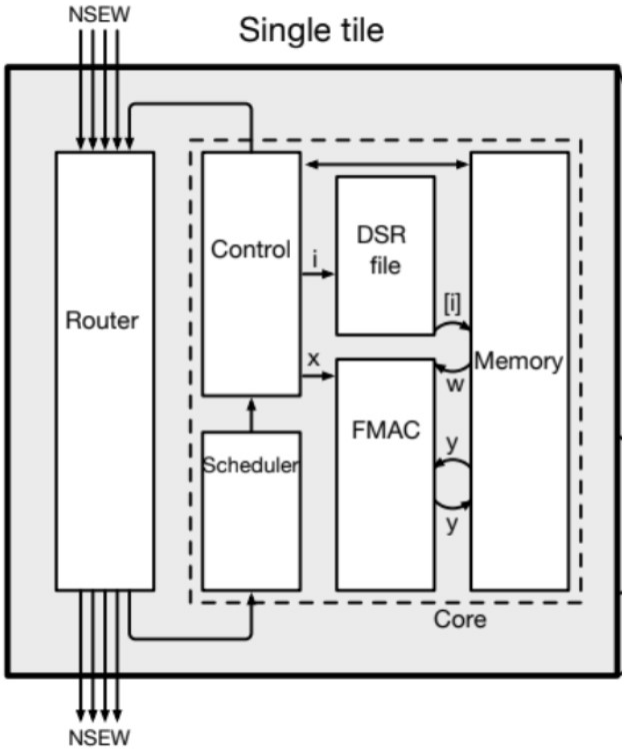

Figure 2.2: The WSE consists of a 2D grid of tiles. Each tile holds a processor, its memory, and a router. The router connects the tile to its North, South, East, and West neighbors.

Each tile holds a modest 48 KB of static random-access memory (SRAM). This novel PE architecture enables fast local coupling of communication with local computation, allowing the CS-2 to move three bytes to and from memory for every floating point operation (FLOP). Compute rate, memory bandwidth, and communication bandwidth are closely comparable on the CS-2, freeing it from the barrier of slow memory access in von Neumann architectures.

Clearly, the CS-2 introduces a new and capable chip layout. The instruction set on the CS-2 is also quite unique. The instruction set can operate on 16-bit integer, 16-bit, and 32-bit floating point types. For 16-bit operands, floating point adds, multiples, and fused multiply accumulates (FMAC) can occur in a 4-way SIMD manner. The instruction set supports SIMD operations across subtensors of four dimensional tensors. The CS-2 includes hardware for tensor addressing, allowing the instructions set to efficiently access tensor data

in memory. This replaces the need for nested loops, eliminating any loop overhead. Tensor operands can have more than four elements and so the instruction can execute for multiple cycles. Instructions with tensor operands can run synchronously or as an asynchronous background thread. There is no context switch overhead. The background thread uses registers and memory as assigned by the programmer or compiler in the instruction. These may not be overwritten until the thread terminates. Any subsequent computations can be delayed until the thread terminates. The core supports nine concurrent threads of execution. Scheduling is performed directly by the hardware, allowing efficient and simultaneous data movement.

	CS-2	IBM Blue Gene
Peak Compute Rate	6.8 PFLOPS	10.06 PFLOPS
PE Count	850,000	786,432
Memory	40 GB	768 TB
System IO Bandwidth	1.2 Tb/s	470 GB/s
Power	17 kW	3.9 MW
Size	15 RU	48 Racks

Table 2.2: The CS-2 compared to the IBM Blue Gene/Q petascale supercomputer located at the Argonne National Laboratory. Although now retired, the IBM Blue Gene/Q ranked number 22 on the Top500 list of supercomputers in November 2019.

The WSE has already shown much promise in tackling large, complex parallel problems. In late 2020, Rocki et al. demonstrated the breakthrough performance on regular finite difference (stencil) problems achieved by the CS-1. They implemented a BiCGStab solver for a linear system arising from the 7-point discretization of a partial differential equation on a 3D mesh. For a 600-by-595-by-1536 mesh, they measured a run time of 28.1 microseconds per iteration. Every iteration required 44 operations per meshpoint resulting in an achieved

performance of 0.86 PFLOPS.[7] From these numbers, we estimate a clock rate of roughly 1 GHz for the CS-1 and assume the same for the CS-2.

In this thesis, we present another application that benefits from the highly parallel, distributed memory, wafer scale architecture. We present a performance model of a Particle In Cell program and compare it to implementations of Particle In Cell on GPU clusters.

CHAPTER 3

PARTICLE IN CELL

Particle In Cell, commonly referred to as PIC, is an algorithm used in plasma physics to simulate particle movement and interaction. PIC simulation is one of the most widely used methods in computational plasma physics. It has been used to successfully study laser-plasma interactions, electron acceleration, and ion heating in the auroral ionosphere, magnetohydrodynamics, magnetic reconnection, and more.[5]

As one may recall from an introductory physics class, charged particles repel or attract each other due to an electric field that they create. Calculating the electric field and forces for a system of two particles is a simple matter of solving a few equations. However, because of its quadratic growth, any system beyond a dozen particles quickly becomes cumbersome. PIC introduces a shortcut − reducing the computation cost by modeling interaction through an electric field, and by localizing the electric field computation into an Eulerian (stationary) grid of mesh cells.

Particles sit within a space that is divided into a grid of cells. The following is a brief overview of the PIC algorithm. A more detailed description of each step is given in Section 5.

1. Find the charge density within each cell by counting how many particles are in the cell and weighing them by their distance from the corner nodes.

2. From the charge density, calculate the electric potential at each node - this involves solving a finite difference equation.

3. Based on the electric potential, compute the electric field.

4. Interpolate the electric field of the cell onto the particle then compute acceleration and update position.

5. Repeat

Due to the locality of computations created by separating the space into cells, PIC can benefit greatly from parallelization. PIC has been previously implemented on GPUs and GPU clusters, offering a significant performance gain compared to CPUs.[8] However, the forced locality introduces an increased need for communication, a stated strength of the CS-2. Calculation of electric potentials rely on values from adjacent nodes, and particle updates rely on the electric fields held at particular nodes. The combination of highly parallel computation, efficient communication, and the demonstrated ability to perform fast stencil computations makes the CS-2 a fitting machine to run PIC codes.

Even so, a key feature of PIC is the ability to achieve parallelism across the mesh as well as the particles. These two separate bases of parallelism present challenges to achieving good scaling when they fail to align spatially. In the next section, we discuss implementation design and mappings that address these challenges.

CHAPTER 4

DESIGN

Mapping this problem onto the CS-2 is not trivial and has a large impact on performance. A key feature of PIC is the ability to achieve parallelism across the mesh as well as the particles. In the ideal case when particle and mesh parallelism align, meaning particle workload and mesh workload are both balanced across the PEs of the machine, the CS-2 will be able to achieve its best performance. We consider potential mappings to achieve this.

We proceed with a straightforward mesh mapping by directly mapping the 2D grid of mesh cells onto the 2D grid of PEs on the CS-2. For simplicity, each PE will be responsible for one mesh cell. An example mapping of a 3-by-3 mesh onto a 3-by-3 grid of PEs is shown in Figure 4.1. This mapping is intuitive for local calculations which, luckily, comprises most of our PIC steps. As we will see later, the Charge Density and Interpolate steps are quite convenient in this case.

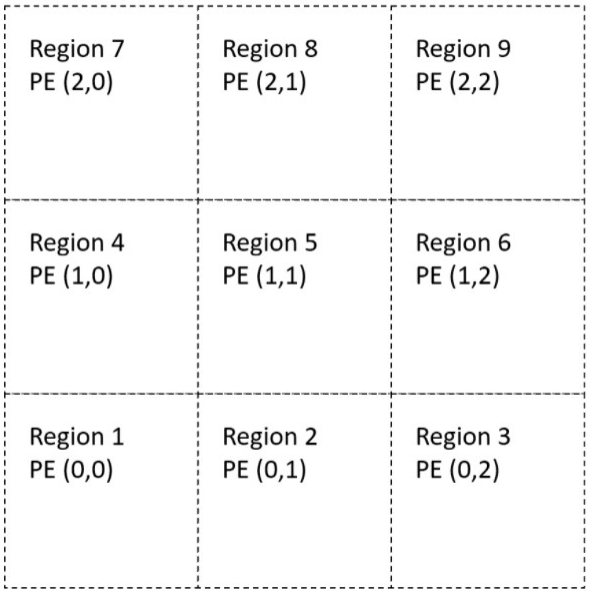

Figure 4.1: A direct mapping of a 3-by-3 grid of mesh cells onto an array of PEs.

Now that we have assigned regions to PEs, we turn to the problem of assigning particles to PEs. Recall that the CS-2 is a distributed memory system, and therefore cannot rely

on some shared memory to keep track of all the particles. Particle data will be distributed amongst the PEs. We will consider two potential methods.

1. Static Particle Method - Similar to how we evenly divided our space to assign to PEs, we can evenly divide our list of particles and assign groups to PEs.

2. Dynamic Particle Method - Somewhat more intuitively, we can assign particles to the PE whose region they are currently located.

We now consider the benefits and drawbacks of these two methods.

4.1 Static Particle Method

The mapping of the Static Particle Method is consistent with our region mapping. It divides the particles up evenly and assigns them to the PEs. Parallelism across the mesh and parallelism across the particle are both achieved. This would help ensure an even distribution of workload. However, unlike regions, particles are dynamic. This means that in order to update particle positions, the PE that owns the particle would need to locate and communicate with the PE that is responsible for the particle's region. This would strain the communication fabric since at every timestep and for every particle on every PE, two PEs that are potentially very far away from each other would need to communicate. The added communication work for each particle would cause the performance of our program to scale poorly with more particles and more mesh cells. In addition, charge density would also be difficult to calculate since PEs would need to perform an all-to-all communication in order to account for all particles in its region.

4.2 Dynamic Particle Method

The Dynamic Particle Method is more intuitive in the sense that as the particles move around in the 2D mesh, the particle data follows by moving around in our 2D grid of PEs.

This method avoids the expensive communication present in the Static Particle Method. Particle acceleration now becomes a local computation. For each particle in its region, the PE computes acceleration based on the electric field of its region and the position of the particle within its region. Both of these are conveniently stored locally.

However, particles will occasionally move out of its PE's region. In this case, the PE will need to transfer the particle's data to the PE responsible for the new region. Evidently, some amount of communication is still necessary, although certainly not as much as the Static Particle Method requires. It is not unreasonable to assume that the timestep is small enough that no particle will move farther than one region away from its last location. In other words, a particle will only ever move over to an adjacent region, never further. A PE will only ever have to transfer data to a direct neighbor, decreasing communication latency.

The movement of particles also creates the additional complexity of finding a data structure that will allow PEs to efficiently add and remove particles from its local list. Data structure type and efficiency will impact the performance of the program. A linked list would be a good candidate for this. In our performance model, we assume the use of an efficient data structure.

A larger problem this method faces is imbalance of workload across PEs. PEs with more particles in its region will have to do more work, leaving other PEs to sit idle for some time. This is an example of when particle and mesh parallelism do not align. There are load balancing tactics that can be used to alleviate this problem. Overloaded PEs can pass some of its workload off to a nearby idle PE. This would improve our rate of performance, but add overhead in workload management. Another possible tactic is to pause the program when a certain threshold of imbalance is reached, and re-partition the grid into a non-uniform Cartesian grid with more density in high-concentration particle regions. In other words, regions with higher particle density will be further divided and spread out onto more PEs, balancing the workload. Rate of performance would benefit, but a large overhead would

be added every time the program halts to re-partition the grid. Workload sharing and re-partitioning will not be considered in our performance model, but may be an interesting direction for future work.

There is also the concern of overloading the memory on a PE. PEs only have 48KB of SRAM to work with. In order to saturate the CS-2, we will certainly be working with more than 48KB worth of particle data. It is possible that particles will congregate on a single PE, and overload its memory. To address this we will need to be conservative with the number of particles we can put on the CS-2. An estimate on maximum problem size is discussed in Section 6.

Lastly, the Dynamic Particle Method restricts the types of boundary conditions we can have. It would be very inefficient to implement a periodic boundary (one where particles that move past the edge of the space wrap around and appear on the opposite side). This is because when a particle wraps around, the PE would have to transfer the particle data across the PE mesh. For this reason, the Dynamic Particle Method is not a good fit for problems that utilize a periodic boundary condition. Luckily, periodic boundaries are not commonly used in particle physics simulations, because they simulate a sort of infinite, repeating space, not often present in the real world.

Ultimately, we believe that the Dynamic Particle Method will produce a better performance. The communication overhead required by the Static Particle Method is not worth its simplicity. In the next section, we move on to an analysis of PIC using the Dynamic Particle Method.

CHAPTER 5

PARTICLE IN CELL ON CS-2

In this section, we analyze a theoretical performance model accounting for potential latencies and using peak computation rates achievable on the CS-2. In addition, we present a partial implementation of PIC on the CS-2 to provide an empirical, but currently imperfect, performance model.

5.1 Parameters

The performance of any PIC program depends mainly on the number of particles and the mesh size. In our performance model we will be considering an input size of N particles on an Mx-by-My mesh. Mapping PIC onto the CS-2, we will only need to consider Np, the number of particles on a specified PE, and the Px-by-Py array of PEs we will be working with. In the ideal case where PEs have perfectly balanced workloads, we will have Np = NPx * Py. Unless otherwise specified we will use a mesh size of Mx*My = Px*Py = 850,000, which is the number of PEs on the CS-2.

Other relevant input information include the timestep (time that passes between each iteration), number of iterations, and space dimensions (area of space particles are able to occupy). We will denote these parameters as dt, I, and X-by-Y respectively. Although these inputs are important in a full run of PIC, they will rarely come up in our performance model since we will only be measuring a single iteration and scale up from there.

Let us also establish the representation of a particle in memory. We will represent a particle as a struct detailing the particle ID, the particle's current position, and the particle's current velocity.

Input Parameter	Denoted By
Number of Particles	N
Mesh Size	Mx-by-My
Number of Particles on PE	Np
Array of PEs on CS-2	Px-by-Py
Timestep	dt
Number of Iterations	I
Space Dimension	X-by-Y

Table 5.1: A list of relevant input parameters the notation used to denote them.

```
struct Particle
{
     int id;
     float position[2];
     float velocity[2];
}
```

Figure 5.1: Particle represented as a struct in C. Particle.position[0] represents the x-coordinate of the particle, and Particle.position[1] represents the y-coordinate. Particle.velocity can be interpreted analogously.

5.2 Preprocessing

As decided in Section 4, we will adopt the Dynamic Particle Method. Given a list of particles as our input parameter, we must create Px*Py lists to load onto the CS-2. Each PE will receive a reduced list consisting of particles residing in its region. Positions of particles on this list must also be offset by the PE's lower left boundary. This makes the position of the particle relative to the PE, and is necessary in order to avoid the PE needing to reference its own location on the WSE. This saves computation in the Charge Density and Interpolation steps.

Now we are ready to load the preprocessed input data onto the CS-2.

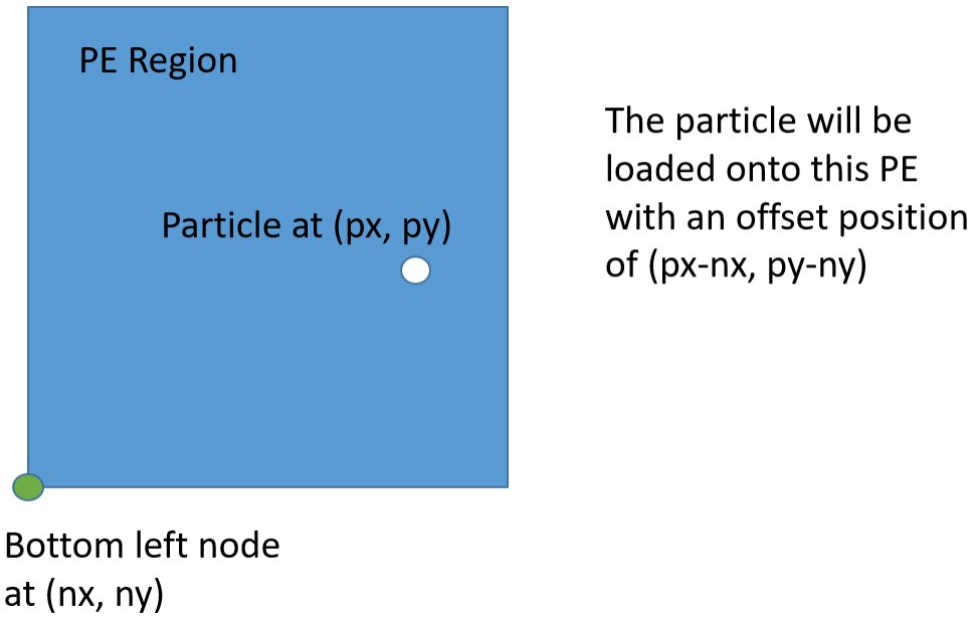

Figure 5.2: The blue square represents the PE's region. The white circle represents a particle with a global location of (px, py). The green circle represents the bottom left node of the PE, located globally at (nx, ny). In preprocessing, we will calculate the offset position as (px-nx, py-nx) so that the coordinates are relative to the bottom left node of the PE.

5.3 Charge Density

The charge density at each node is calculated as the sum of all charges in adjacent cells weighted by their distance from the node. Each PE is responsible for a rectangular region with four corner nodes. For example, the green node in the bottom left corner of the PE region in Figure 5.3 would receive a contribution from the particle (colored white) equal to its charge multiplied by AG divided by the area of the entire region, where $A_G = (Sx - x) * (Sy - y)$. The purple, red, and yellow nodes would receive a contribution from the white particle calculated analogously with A_P, A_R, and A_O respectively. Notice that nodes closer to the particle receive a more heavily weighted contribution, and nodes farther away receive a lighter contribution. In addition, the sum of all four contributions produced by a particle is equal to its charge.

The PE calculates and sums the contribution for every particle in its region for each of

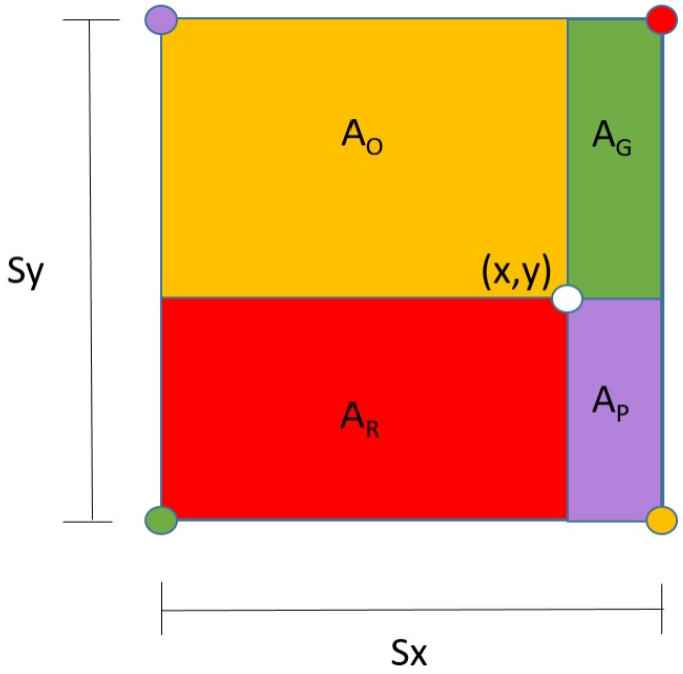

Figure 5.3: The blue square represents the PE's region. The white circle represents a particle with a global location of (px, py). The green circle represents the bottom left node of the PE, located globally at (nx, ny). In preprocessing, we will calculate the offset position as (px-nx, py-nx) so that the coordinates are relative to the bottom left node of the PE.

the four corner nodes. Because these corners overlap with adjacent PEs, we then have to add up the contributions of the four adjacent PEs to arrive at the final charge density of the node. This can be done in post processing since the data will have to be reloaded in between this step and the next.

In total, this requires 10 operations per particle. Theoretically, with proper data access patterns and thoughtful use of FMAC operations, we should be able to achieve 4 operations per cycle. We would then be able to complete this step in 2.5*Np cycles.

Our implementation of this step does not quite reach this performance. Running the program on a CS-2 simulator showed that we are able to complete this step in 10*Np cycles, achieving only 1 operation per cycle per PE. The culprit of the inefficiency is in the accumulation steps. Since we are summing into a register, we do not receive the speed up that tensor operations allow. This slows our program down due to the need to read and write

into the same register Np times. However, there are likely more efficient ways to implement this step using FMAC operations that we have yet to explore. Therefore, we present this implementation only as an empirical example and will continue considering the theoretical performance of 2.5*Np cycles.

5.4 Finite Difference

Using the charge densities, we can compute the electric potential at each node. This is done by solving the following system of equations for $\phi_{i,j}$, the electric potential at node (i,j). Note that x represents the width of the mesh cell, y represents the height of the mesh cells, n_0 represents the average charge density, and $n_{i,j}$ represents the charge density at node (i,j).

$$\frac{\phi_{i-1,j} + 2\phi_{i,j} + \phi_{i+1,j}}{\Delta^2 x} + \frac{\phi_{i,j-1} + 2\phi_{i,j} + \phi_{i,j+1}}{\Delta^2 y} = n_{i,j} - n_0$$

To solve this differential equation, we use a finite-difference method. We approximate the performance of a 2D iterative solver on the CS-2 by implementing a stencil program that performs a single iteration. This program performs 9 operations. Unfortunately, the iterative nature of this program means it will at most be able to perform one operation per cycle since it must wait for input from other PEs. Thus it will require 9 cycles plus 11 cycles of communication latency. With theoretical peak performance, this step should take 20 cycles. However, our implementation of this step runs in 30 cycles.

We estimate that iterative solvers take less than 50 iterations to converge. This is supported by estimates made by Marjanovic et al. at the High Performance Computing Center Stuttgart.[6] Although a rough approximation, this gives us a generous estimate of 800 operations and 1,500 cycles to complete this step.

5.5 Electric Field

Now that we have the electric potential at each node, calculating the electric field is relatively simple. At every node, we solve the following two equations:

$$E_{x,i} = -\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta x} \qquad E_{y,j} = -\frac{\phi_{i,j+1} - \phi_{i,j-1}}{2\Delta y}$$

We implemented this by loading the electric potential at each node onto its corresponding PE. The PEs then communicate with its direct North and South neighbors to receive $\phi_{i,j+1}$ and $\phi_{i,j-1}$ to calculate E_y, and East and West neighbors to receive $\phi_{i+1,j}$ and $\phi_{i-1,j}$ to calculate E_x. This gives us the electric field at each node.

The theoretical performance and measured performance is the same for this step. We implemented this step with 5 operations. It takes 14 cycles to run. The bulk of the runtime of this program is consumed by communication latency.

5.6 Interpolate

To find the electric field at a particle's location, we interpolate the electric fields at the corner nodes of its region. This is often referred to as a bilinear interpolation. This step is essentially the inverse of the Charge Density step. Looking again at Figure 5.2, the green node will have a contribution to the electric field at the particle equal to its electric field multiplied by A_G divided by the area of the entire region, where $A_G = (Sx - x) * (Sy - y)$. The white particle would receive a contribution from the purple, red, and yellow nodes calculated analogously with A_P, A_R, and A_O respectively.

This step requires 23 operations per particle to perform. With peak performance and using FMAC operations, we could theoretically complete this step in 3.75*Np cycles. Our implementation of this program completes in approximately 6*Np cycles, achieving between 2 to 4 operations per cycle. Despite requiring more operations, this program is more efficient

than the Charge Density program since we are no longer accumulating Np times into a single register. We now only read from that register Np times and write into Np particles structs. However, our implementation suffers from inefficient memory bank access patterns.

5.7 Particle Update

The particle update step is quite straightforward. Given the acceleration of each particle, we update position and velocity according to these elementary physics equations:

$$v_f = v_0 + at \qquad x_f = x_0 + vt$$

This step takes 8 operations per particle: 2 operations each to update x-velocity, y-velocity, x-position, and y-position. On this program, the CS-2 is able to achieve 8 operations per cycle, the theoretical peak performance, using FMAC. We complete this step in Np cycles.

CHAPTER 6

PERFORMANCE MODEL AND RESULTS

6.1 Problem Size

First, it is productive to discuss the problem size that will saturate the CS-2 without overloading its memory. The CS-2 holds 850,000 PEs, in a rectangular grid. Our analysis assumes that each PE corresponds to one mesh cell. This means that the finest mesh we can support is a 900 by 900 grid on the CS-2. It is possible to restructure the mapping so that PEs can handle multiple mesh cells, or even a column of mesh cells in a 3D PIC implementation. This is further discussed in Section 7.

Each of the 850,000 PEs on the CS-2 has 48KB of on-chip memory. While this makes memory latency very low, it severely limits the size of problems we can put on the CS-2. Just how limited is our problem size? The piece of computation that is most memory intensive is the particle update step where we update particle position and velocity based on acceleration. In this step, we need to keep a particle ID, x-coordinate, y-coordinate, x-velocity, and y-velocity for each particle. Assuming each piece of data is one word (16 bits), we need to store 10 bytes per particle. 10 bytes per particle and a maximum of 48KB on the PE means that we can have a maximum of 4800 particles on any single PE. The CS-2 has approximately 850,000 PEs, bringing us to a total of 4.08 billion particles over the entire chip.

In order to avoid overloading any single PE due to particle congregations, we need to restrict these numbers even further. We will reduce the number of particles loaded on a single PE at the start of a simulation to 10% of the calculated maximum. This brings us down to 480 particles on any single PE and 408 million particles across the entire chip.

Step	Number of Operations per PE
Charge Density	10*Np
Finite Difference	400
Electric Field	5
Interpolate	23*Np
Particle Update	8*Np
Total Work per PE	41*Np + 405

Table 6.1: Number of Operations per PE by step.

6.2 Performance Model

Putting all the steps together, we approximate that a PIC implementation on the CS-2 will require $41 * Np + 405$ operations per PE, or $(41 * Np + 405) * (850,000)$ across the entire system. Theoretically, this program could run in $7.25 * Np + 1014$ cycles on the CS-2. However, our implementation takes $17 * Np + 1514$ cycles, not including preprocessing or intermediate setup between steps. Using our estimate of a 1 GHz clock rate, the total runtime of this PIC program will be $(7.25 * Np + 1014)$ billion seconds and $(17 * Np + 1514)$ billion seconds in the theoretical and observed performance model, respectively. This gives us a performance rate of $((41 * Np + 405) * 850000)/(7.25 * Np + 1014)$ GFLOPS and $((41 * Np + 405) * 850000)/(17 * Np + 1514)$ GFLOPS, respectively. These numbers are better summarized in Table 6.3. Graph 6.1 shows the theoretical and observed performance given a range of problem sizes. The CS-2 performs better with larger problem sizes because it is able to utilize more of its compute power. On our maximum problem size of 408 million particles, the CS-2 can achieve 3.666 PFLOPS in our theoretical performance model and 1.723 PFLOPS in our observed performance model.

Step	CS-2 Theoretical Latency	CS-2 Observed Latency
Charge Density	2.5 * Np cycles	10 * Np cycles
Finite Difference	20 cycles * 50 iterations	30 cycles * 50 iterations
Electric Field	14 cycles	14 cycles
Interpolate	3.75*Np cycles	6 * Np cycles
Particle Update	Np cycles	Np cycles
Total	7.25*Np + 1014 cycles	17*Np + 1514 cycles

Table 6.2: Number of cycles per PE needed to complete each step.

	CS-2 Theoretical Summary	CS-2 Observed Summary
Total Work per PE	41*Np + 405	41*Np + 405
Total Work Across System	(41*Np + 405) * 850000	(41*Np + 405) * 850000
Total Number of Cycles	7.25*Np + 1014	17*Np + 1514
Total Runtime (Seconds)	(7.25*Np + 1014) Billion	(17*Np + 1514) Billion
Rate of Performance (GFLOPS)	$\frac{(41*Np + 405) * 850000}{7.25*Np + 1014}$	$\frac{(41*Np + 405) * 850000}{17*Np + 1514}$

Table 6.3: Performance Summary of PIC program on the CS-2. Assumes all 850000 PEs are utilized.

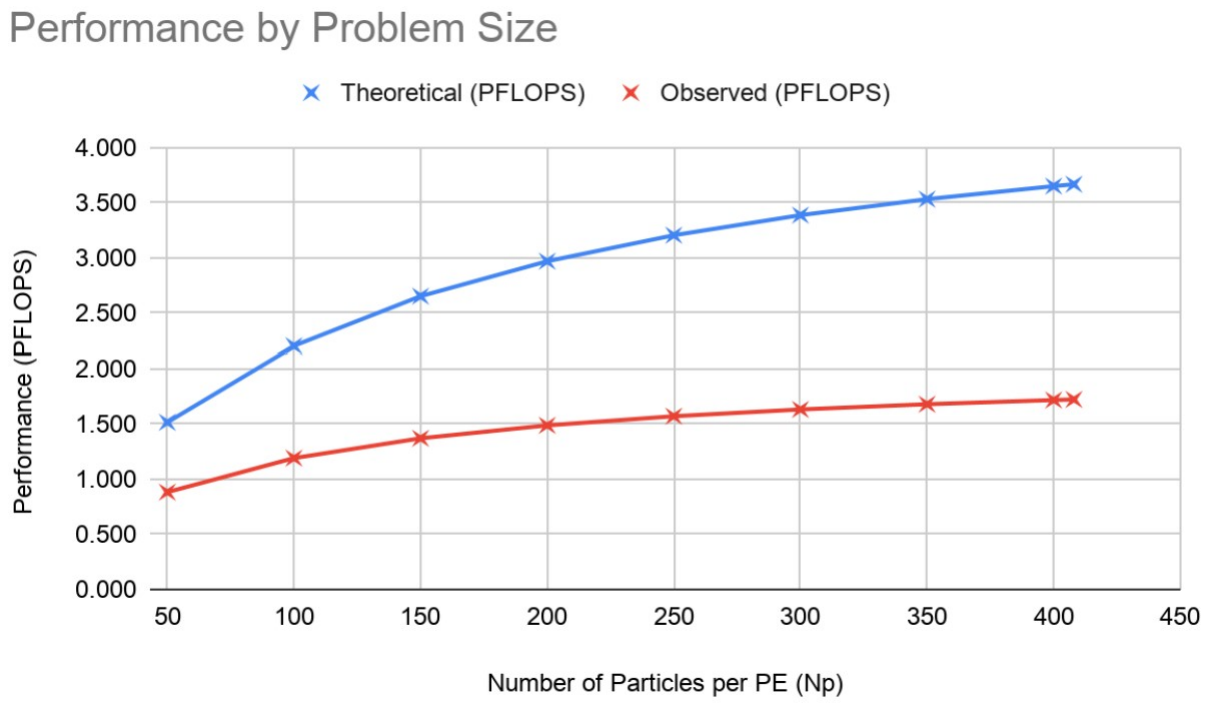

Figure 6.1: Plot of theoretical and observed performance based on problem size. Np = 408 is the maximum problem size.

CHAPTER 7

DISCUSSION

7.1 Additional Complications

7.1.1 Load Imbalance

There are some additional complications that are worth addressing, the first of which is load imbalance. It is usually unnatural for particles to congregate. In an open space, particles will repel and attract each other in such a way that causes them to spread out, not congregate. Regardless, it is unlikely that we will have a perfect balance of particles between PEs at every iteration. Load imbalance is an important factor in determining valid input sizes that reduce the probability of overloading PE memory as well as performance. The performance of the CS-2 is only as fast as the slowest PE. In the case of PIC, the slowest PE is the PE with the most particles in its region. Graph 7.1, shows how load imbalance can have a negative impact on performance.

7.1.2 Particle Data Transfers

The second complication we consider is the possibility of particles moving across PE boundaries. When this happens, the particle data must be transferred to the PE whose region it is now in. Typically, with a reasonably small timestep, particles will not move farther than one cell size in one iteration. In fact, with a small enough timestep, particle movement across boundaries is not likely at all. However, if a particle does move out of its current region, it will end up in one of the adjacent regions, and the particle's data will only need to be transferred to one of the neighboring PEs.

A program that performs this particle data transfer would first have to check whether the particle's position is out of the PE's boundaries in the north, east, west, or south directions,

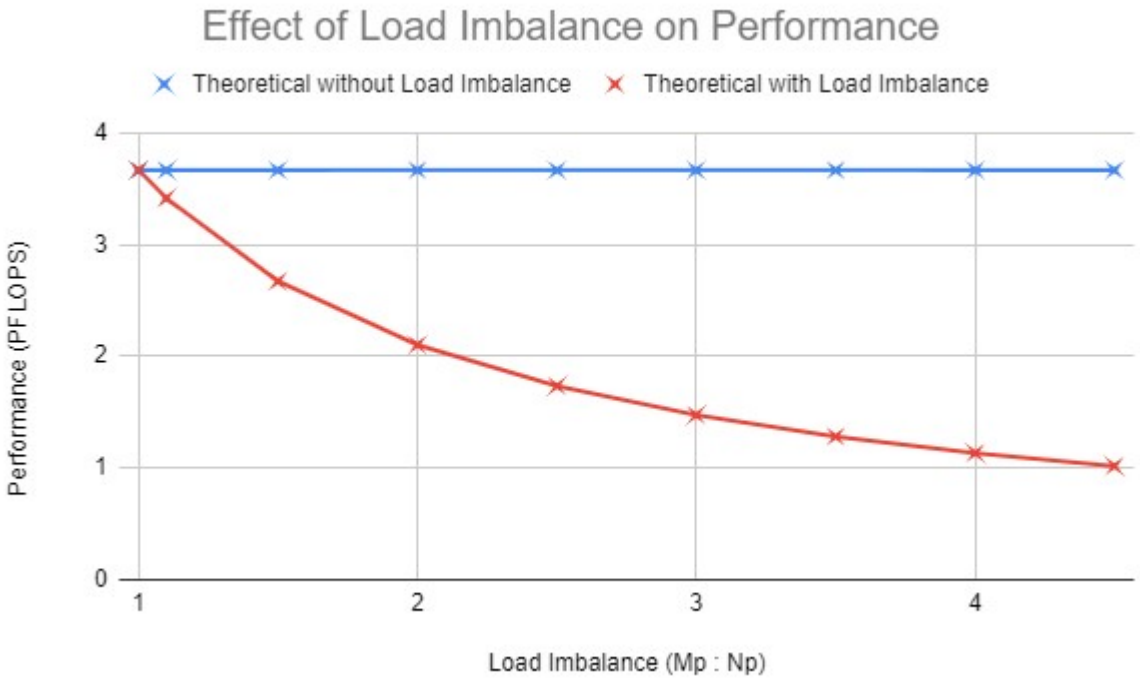

Figure 7.1: Load imbalance can have a dramatic impact on performance. This graph compares the theoretical performance given different ratios of load imbalance. Mp = maximum number of particles on any given PE. Np = Average number of particles per PE.

and if it is, send the particle's data to the appropriate PE. In our implementation of this step, it takes 14 operations to check a particle's position, and 5 operations to send a particle data out if necessary. The extra 5 operations it takes to send particle data out is negated by the fact that particles that are out of bounds will often not need to complete the 14 operations of conditional checks. Once one of the conditionals is satisfied, the PE will send the data out without finishing the rest of the conditional checks. Therefore we estimate that this step will take 14*Np operations overall. Our run of this program takes 20 cycles per particle.

In addition, once a particle's data is transferred out of the PE's memory, it must be "removed" to avoid performing future updates on it. The naive method would be to simply mark the particle by zeroing out the particle ID. Any new particles transferred into the region would simply be appended to the end of the PE's particle list. This is an inefficient use of

space, and many clever data structures such as linked lists would be better candidates. However, due to the low-level programming required to work with the CS-2, we prefer a method that is simple to model. To avoid filling up our particle list with null particles and running out of space, we introduce a condense step.

In the condense step, the PE shifts the next "active" particle's data into the last zeroed-out particle space. Using a two pointer method, the PE would check if the particle ID is zeroed-out, and if so, place a pointer there. With the second pointer, continue along the particle list until it reaches an "active" particle. Then move the active particle data into the first pointer location. In the worst case, the PE moves every piece of particle data once, and eight operations will be needed per particle (one conditional operation and one move operation per piece of particle data). Taking advantage of the CS-2's SIMD architecture allowing it to move four words per cycle, we estimate that this step will take three cycles per particle.

How often we need to perform a condense step depends on how quickly PEs fill up their particle lists. We settle for an overestimate and perform a condense step at every iteration.

Factoring these complications in, we update our performance model. Instead of taking the average number of particles (Np), we instead take the maximum number of particles on any given PE (Mp).

7.2 Mapping Modifications

We began the formulation of our performance model with the assumption that each PE will be assigned one mesh cell. This restricts the size and shape of the mesh to the number and arrangement of PEs on the CS-2. There are modifications that can be made to accommodate larger mesh sizes.

The first modification is blocking. We can place a rectangular array (or block) of mesh cells onto a single PE. The overall layout of the mapping would be preserved. However, it

Step	Number of Operations	Number of Cycles	Performance Rate (GFLOPS)
Standard PIC	(41*Np + 405) * 850000	7.25*Mp + 1014	$\frac{(41*Np + 405) * 850000}{7.25*Mp + 1014}$
Particle Data Transfer	14*Np * 850000	20*Mp	
Condense	8*Np * 850000	3*Mp	
Total	(63*Np + 405) * 850000	30.25*Mp + 1014	$\frac{(63*Np + 405) * 850000}{30.25*Mp + 1014}$

Table 7.1: Updated PIC performance model taking into account load imbalance and particle data transfer.

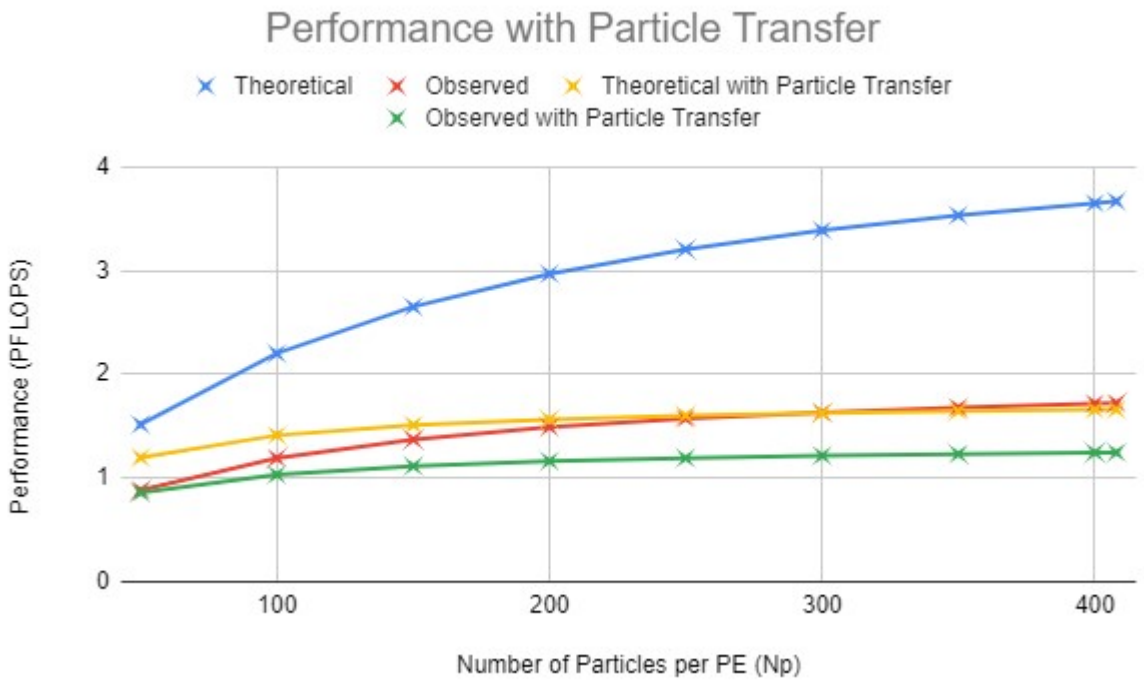

Figure 7.2: PIC Performance after accounting for particle movement.

would impact the PE's workload. Suppose we use blocks of size B-by-B. The maximum number of particles would remain the same. The Charge Density step would gain added complexity with more nodes to keep track of. However, the overall number of operations and latency would remain the same since it relies solely on the number of particles on the PE. The Finite Difference step would acquire additional communication latency. Each PE would have to communicate $4 * B$ values and receive $4 * B$ values instead of just 4 each. The number of operations performed on each PE would also increase by a factor of B^2 (the number of mesh cells on the PE). Assuming we can achieve 4 operations per cycle, this would imply an increase in latency by a factor of $2 * B + 0.25 * B^2$ for this step. The Electric Field step would receive the same increase in work and latency. Similar to the Charge Density step, the Interpolation and Particle Update steps' performance would remain unchanged. Table 7.2 shows the performance result of blocking.

The second modification we discuss is tessellation. Similar to blocking, tessellation places multiple mesh cells on a single PE. However, instead of a continuous block of mesh cells, we "fold" the mesh onto the PEs, achieving continuity across PEs, but not within PEs. This "folding" is better described in a diagram. Figure 7.1 shows a tessellation of a 9-by-9 mesh grid onto a 3-by-3 array of PEs. Tessellation, like blocking, would have no effect on the work or latency of the Charge Density, Interpolation, or Particle Update steps. However, tessellation would require B^2 times more values communicated in both the Finite Difference and Electric Field steps, as well as B^2 more operations to perform. Assuming a performance of 4 operations per cycle, this would imply an increase in latency by a factor of $0.5 * B^2$. Figure 7.3 shows how performance is affected by the block size, B.

7.3 Comparing Against Other Machines

First, we isolate the Finite Difference step to compare its performance to more traditional HPCG code running on a CPU. We would like to see a performance improvement propor-

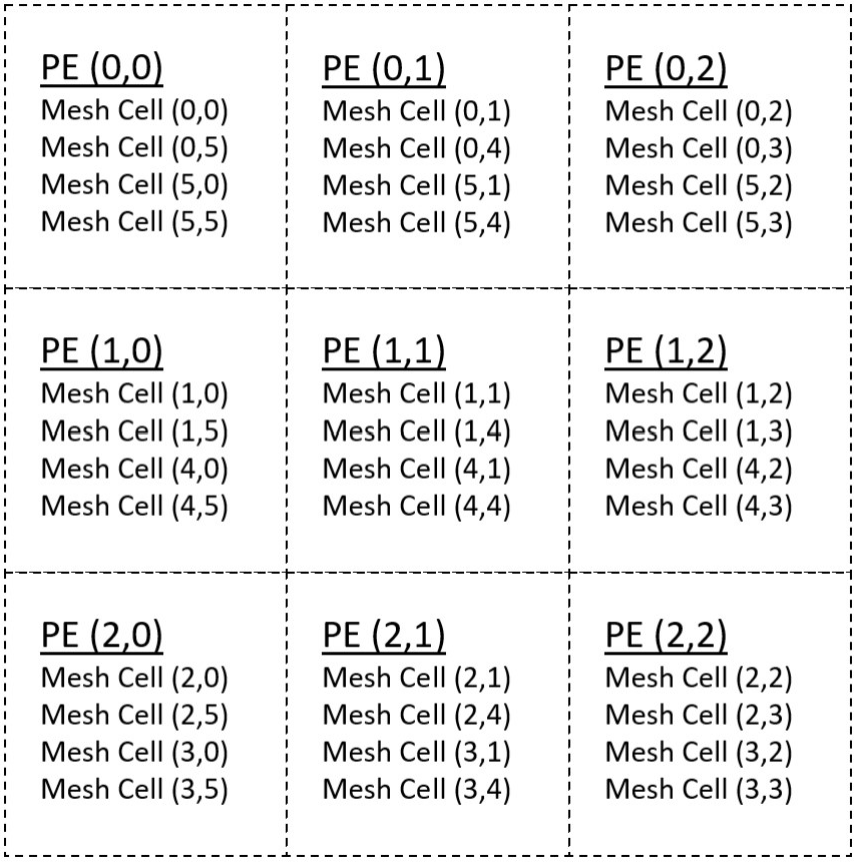

Figure 7.3: A 9-by-9 grid of mesh cells tessellated onto a 3-by-3 grid of PEs. The mesh grid is "folded" onto the array of PEs. This mapping preserves adjacencies of mesh cells, keeping communication patterns consistent with non-tessellated mappings.

Step	Original	Blocking	Tessellation
Total Work per PE	(41*Np + 405) * 850000	(41*Np + 405*B^2) * 850000	(41*Np + 405*B^2) * 850000
Total Latency	7.25*Np + 1014	7.25*Np + 1014*(2*B + 0.25*B^2)	7.25*Np + 1014*B^2

Table 7.2: Compares the Total Work and Total Latency between the original mapping, block mapping, and tessellation mapping.

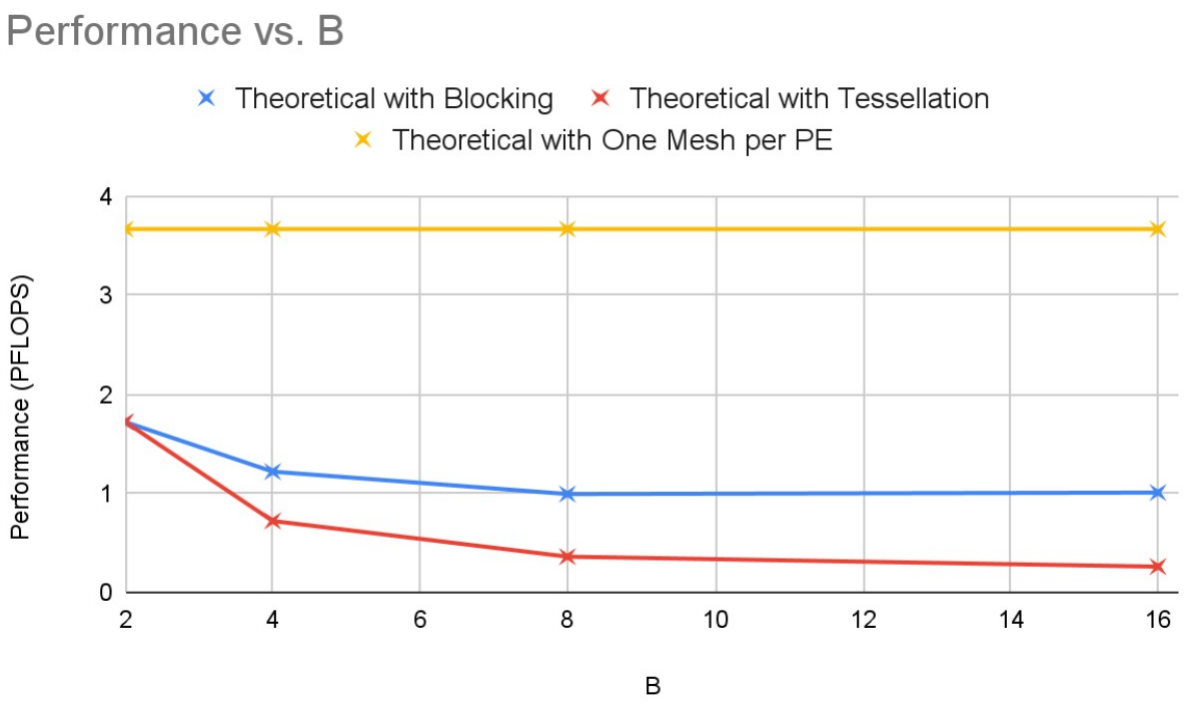

Figure 7.4: Performance worsens with larger B. Blocking produces better performance than tessellation. In all cases, blocking and tessellation perform worse than the original one mesh per PE mapping. Thus, we recommend only using blocking or tessellation when the mesh size is larger than 850,000.

tional to the number of PEs, but anticipate something closer to 1000 times. Using the HPCG benchmark program provided by the Innovative Computing Laboratory at the University of Tennessee[2], we benchmarked the performance of an Intel Core i7-8700K CPU with 6 cores running at 3.7 GHz. We chose a problem size consisting of a 208-by-312-by-208 matrix, ensuring that it would fit nicely into 16 GB of memory. This program ran in 70.0071 seconds, performing 250.961 billion floating point operations, resulting in a performance rate of 3.53 GFLOPs per second.

The CS-2 performs the Finite Difference step with this problem size in 1500 cycles, performing 680 million floating point operations, resulting in a performance rate of 453.3 TFLOPs per second. This is a somewhat unsatisfying comparison seeing as the HPCG benchmark implements a 3D iterative solver running on a 6 core CPU while our model is of a 2D iterative solver running on 850,000 PEs. However, it is difficult to find or access

machines that can fairly compare to the CS-2.

A common, and widely studied, parallel device is a GPU. Like the CS-2, GPUs are a highly parallel accelerator. Unlike the CS-2, however, GPUs utilize off-chip, shared memory. GPUs also lack the wafer scale aspect that differentiates the CS-2 from almost all other machines and delivers its unmatched communication speeds. With this said, GPUs are much more common-place than CS-2s currently are and have inspired many PIC implementations.

We compare our performance model to a PIC implementation on the Kepler GPU architecture by Shah et al. From a run of their PIC implementation, they measured the Nvidia Tesla K40, GTX 690, and the Quadro K620 to perform 1.464 TFLOPs, 117 GFLOPs, and 25 GFLOPs per second respectively on an input size 5.24 million particles on a 512-by-512 mesh.[8] The CS-2 outperforms the Tesla K40 with a 1000x speed up. Again, this is not an entirely fair comparison. The CS-2 costs much more than a Tesla K40, and is much larger with more PEs. However, it is important to note that performance of GPU clusters do not scale proportionally to the number of GPUs in the system. Doubling the number of GPUs used will not double the performance. For this reason, the CS-2 as a whole system will perform better than a GPU cluster with a comparable processor count.

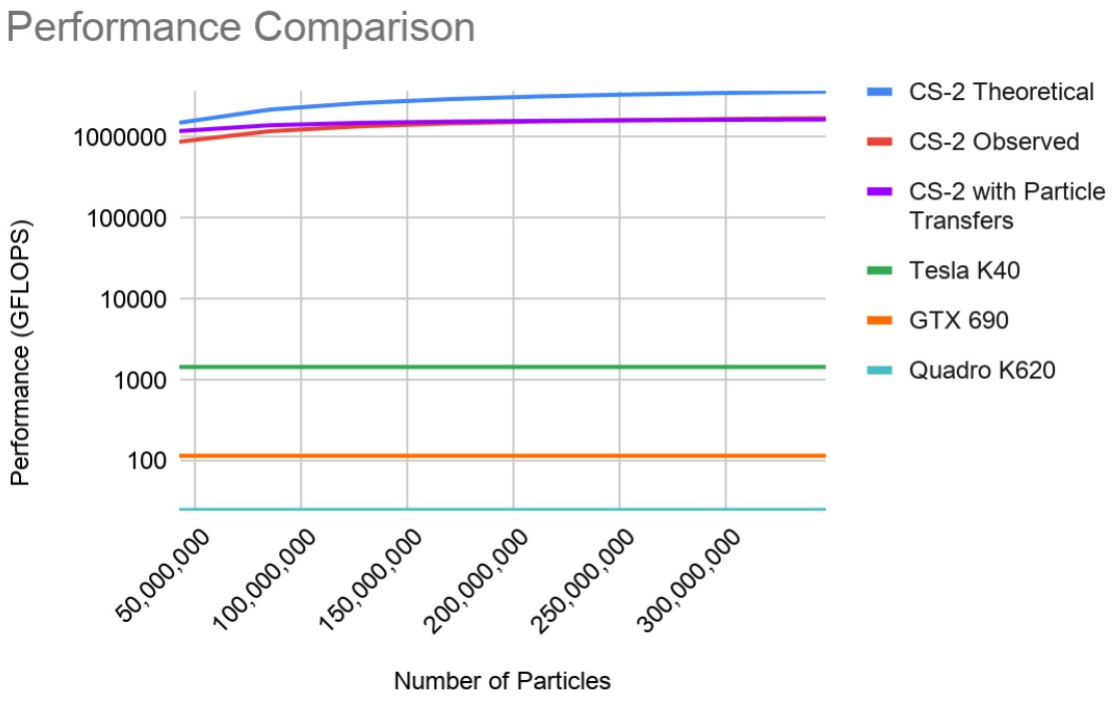

Figure 7.5: A log scale graph comparing the performance of the Tesla K40, GTX 90, and Quadro K620 to the CS-2.

CHAPTER 8

CONCLUSION

In this thesis we explored the viability of PIC codes on the CS-2. We considered possible ways to map the problem onto the machine, and presented theoretical performance models. We also partially implemented PIC on the CS-2 to get empirical performance results. At the maximum problem size of 408 million particles and 850,000 mesh cells, the CS-2 has the potential to achieve 3.67 PFLOPS. In our implementation, we estimate a performance of 1.72 PFLOPS on the same problem size. Because this is only a partial implementation, it is worthwhile to consider both the empirical and theoretical numbers due to potential performance improvements in future implementations. We also account for potential complications such as load imbalance and particle movement across PE regions. Factoring this in and estimating the additional latency of a particle transfer step, we estimate a performance of 1.66 PFLOPS. A decrease in performance from our initial theoretical estimates, however still dominating in comparisons to other machines such as CPUs and GPUs.

The particle implementation and empirical results we gathered showcase the potential of the CS-2 beyond theoretical projections. It sets the baseline for future implementations. If the numbers presented in this paper aim to accomplish anything, it is to inspire future implementations and investigations into PIC on the CS-2. The design and mapping considerations are essential elements to any PIC implementation on the CS-2 and will serve future research on this topic well.

Clearly, a complete implementation of PIC with attention to efficiency will allow for more concrete performance data that is more readily comparable to other PIC performance models and solidify the strength of the CS-2 on PIC computations. A feature of a more efficient implementation may include workload sharing between PEs to mitigate latency introduced by workload imbalance. We strongly encourage future research to pick up where this thesis ends.

In addition, difficulty presented itself in the pursuit for machines that can be fairly compared to the CS-2, a high-end and novel piece of technology. Any machine that might be comparable to the CS-2 is, like the CS-2, expensive and difficult to access, and as of yet, have not seen any published PIC implementations. If better comparisons can be found, this would strengthen the argument that the CS-2's impressive performance on PIC is due to its unique architecture rather than a result of expensive machinery.

REFERENCES

[1] Chen, F., and Shen, J. A gpu parallelized spectral method for elliptic equations in rectangular domains. *Journal of Computational Physics 250* (10 2013), 555–564.

[2] Dongarra, J., Heroux, M., and Luszczek, P. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications 30* (08 2015).

[3] Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. pp. 133–137.

[4] Kindratenko, V. V., Enos, J. J., Shi, G., Showerman, M. T., Arnold, G. W., Stone, J. E., Phillips, J. C., and Hwu, W.-m. Gpu clusters for high-performance computing. In *2009 IEEE International Conference on Cluster Computing and Workshops* (2009), pp. 1–8.

[5] Liu, G., and Liu, M. *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. 01 2003.

[6] Marjanović, V., Gracia, J., and Glass, C. Performance modeling of the hpcg benchmark. pp. 172–192.

[7] Rocki, K., Essendelft, D. V., Sharapov, I., Schreiber, R., Morrison, M., Kibardin, V., Portnoy, A., Dietiker, J. F., Syamlal, M., and James, M. Fast stencil-code computation on a wafer-scale processor, 2020.

[8] Shah, H., Kamaria, S., Markandeya, R., Shah, M., and Chaudhury, B. A novel implementation of 2d3v particle-in-cell (pic) algorithm for kepler gpu architecture. pp. 378–387.