Implementing 3D Jacobi Method in Cell

Md. Kamruzzaman

Abstract
IBM Cell Broadband Engine (CBE) is an interesting architecture that provides amazing performance for floating point (especially single precision) computation intensive applications. Cell also provides very impressive gFlops for double precision operations. 3D Jacobi method is a heavily utilized method in scientific computations. In this project, we implement the parallel version of 3D Jacobi method in Cell. We show that our implementation can exploit the underlying Cell architecture very well, and achieves impressive performance/gFlops for various grid sizes. The current implementation achieves 0.93 gFlops for 1 SPE, 6 gFlops for 8 SPE, and 6.75 gFlops for 16 SPE using double precision operations. We also compare the performance of a CBE with an 8-core system, Intel 64 (Clovertown) 2.33 GHz dual socket quad core and find that the implementation in Cell is 6-7 times faster.

1. Introduction
High packing density allows putting multiple cores in a single chip nowadays. The power factor limits the option of increasing clock speed and so, multicore architectures try to extract more performance by exploiting parallelism. The memory gap is another factor that limits the advantage of high clock speed. On the other hand, from applications’ point of view, the diversity is increasing. Some applications are embarrassingly parallel, some are memory intensive, some require lots of floating point operations, some have regular (irregular) access pattern to data, some have streaming data access, etc. With the current technology, it is almost impossible to build an architecture, which will work well for all possible applications.

IBM Cell [1] is a specialized architecture that is highly suited for gaming applications. Cell boasts of an amazing 25.6 gFlops per SPE for single precision (SP) operations. SP operations are not suitable for most scientific computations, but works pretty well for graphics applications. Cell also provides around 1.83 gFlops per SPE for double precision (DP) operations, which is still very impressive comparing to other architectures. Thus, Cell is a good platform for implementing scientific applications, too. However, it is not easy to program Cell to get good gFlops for several limitations – specially the limited local store, high branch penalty, need for vectorizations, relatively high load/store latency, etc.

For this project, we pick the 3D Jacobi method, which is a frequently used method to solve discrete Poisson equation. The 3D Jacobi method is a memory intensive scientific application that accesses memory in three dimensions. So, for traditional architectures, performance gets very poor if the total grid does not fit in cache. The parallel version also suffers the same problem, when each sub-grid exceeds the cache size. In this project, we implement the parallel version of 3D Jacobi in Cell. The main goals of this project are to explore the Cell architecture, and getting high performance for an application which is memory intensive, and traditional architectures do not work that well.

The remainder of the report is organized as follows. Section 2 gives a brief overview of the Cell architecture, and programming. Section 3 describes the parallelization of 3D Jacobi method. Section 4 presents the basic ideas of implementing Jacobi method in Cell, load distribution over SPEs, vectorization, etc. Section 5 analyzes the performance optimization techniques that we use. Section 6 provides the methodology. Section 7 shows experimental results, and section 8 concludes.
2. Programming Cell Broadband Architecture

In this section, we first give a simple overview of Cell architecture, followed by Cell programming techniques, the hardware details of Cell, which are important for programming Cell, and gFlops calculation for Cell BE. SPEs are the most important part of Cell. So, all of our following discussion is for SPE unless specified otherwise.

2.1. Cell Architecture

Figure 1 shows the high-level block diagram of a Cell BE. Each Cell BE has a 64-bit power core, power processing element (PPE), and eight synergistic processing elements (SPE). The SPEs are the main computation units of Cell and each of them run at 3.2 GHz. SPEs are SIMD units and can work on 16 bytes vector (four SP, or 2 DP). Each SPE has 256K local store (LS), and SPEs cannot access outside their LS. DMA operations bring data from memory to LS, and vice versa. Cell has a high Element Interconnect Bus bandwidth, 204 GB/s, and its integrated memory controller provides a peak raw bandwidth of 25.6 GB/s to an external XDR memory. However, the raw bandwidth reduces to various reasons; effective bandwidth is around 21 GB/s [1]. All SPEs share the 21GB/s bandwidth for their DMA operations to memory.

Each SPE has two pipelines, and there are two groups of instructions. One group executes in pipeline-0 (Even pipe), and the other one executes in pipeline-1 (Odd pipe). Dual issue is possible when there is a fetch group of two instructions (register dependencies already satisfied, no structural hazard), and the first instruction can be executed in even pipeline whereas the second one can be executed in odd pipeline [2].

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Latency (Clock cycles)</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load, Store</td>
<td>6</td>
<td>Odd</td>
</tr>
<tr>
<td>Double Precision operations</td>
<td>13</td>
<td>Even</td>
</tr>
<tr>
<td>Short Integer operations</td>
<td>2</td>
<td>Even</td>
</tr>
<tr>
<td>Shuffle</td>
<td>4</td>
<td>Odd</td>
</tr>
<tr>
<td>Branch resolution</td>
<td>4</td>
<td>Odd</td>
</tr>
</tbody>
</table>

Table 1: Important instructions’ latency and corresponding pipelines

Table-1 shows the important instructions (frequently used in our implementation), their latency, and execution pipeline. For each SIMD double precision operations, there is a six cycles stall, and no instructions of any type can issue at that time [2]. From Table-1, we see accessing local store is expensive comparing to traditional L-1 cache (1-2 cycles latency). SPEs have very large register file, 128x128 bit,
and effective use of this register file greatly reduces access to LS. SPEs have no branch predictor, and branch mispredict penalty is 18-19 cycles.

2.1. Programming considerations

Programming Cell is difficult because of limited local store. In traditional architecture, memory operations can access any location in application’s address space. Hardware handles all the complexities of accessing memory hierarchy (L1-cache, L2-cache, memory, etc.). The LRU replacement policy of caches also works pretty well to keep working set in caches. However, in Cell, SPEs cannot access outside its local store. The general idea to program SPE is to bring data in LS beforehand, and work on that. This scheme works well for applications with regular pattern of data access, but programs with irregular data access (graph algorithms, linked lists, etc.) will be hard to implement on Cell. Even for programs with regular pattern of data access, it requires careful design to reduce unnecessary DMA operations and to use LS optimally.

Using large register file is crucial in Cell. Judicial use of registers avoids expensive accesses to LS. Large register file is especially helpful for loop unrolling. In many cases, consecutive loop iterations share data and registers can store some part of that data.

Data dependency restricts instruction issue, and so it is important to ensure that there are independent instructions always. Loop unrolling plays an important role here, as there can be data dependency in an iteration of a loop, but no true dependency (read after write) among different iterations. Also changing the computation order can remove unnecessary data dependency.

Dual cycle issue should be exploited. Compilers try to do that, but sometimes manual rescheduling of instructions gives better dual cycle efficiency. To achieve dual cycle issue, it requires instructions for both odd and even pipes and so instruction selection is a key thing. For example, latency of integer instructions can be avoided by dual piping with load/shuffle operations. 32-bit integer operations are costly in SPE. Therefore, whenever possible, it is better to use 16-bit integers. For frequently executed branches, branch hints reduce the overall impact of branch penalty.

Vectorization is important to get good performance. Data should be 16-byte aligned for vectorization. DMA operations also require 16-byte alignment (preferably 128-byte). If alignment error occurs, programs crash and give “bus error” without any other information. “Bus error” also occurs, when two DMA operations want to write same memory locations. Each DMA operation can transfer maximum 16K, and the transfer size needs to be a multiple of 16. Applications should use less large DMA transfers, rather than more small DMA transfers.

It is important to consider all the above things while designing algorithm for Cell. The best algorithm for traditional architectures may not be the best for Cell. A complete redesign of algorithm is important to get good performance from Cell architecture. Our initial code used only 10 lines for the inner most loop, but that achieved only 20% of peak gFlops for one SPE. The new implementation uses around 220 lines, and achieve over 90% of peak gFlops.

2.1. Peak gFlops

In Cell, one can easily be befuddled about getting good gFlops. The advertised 25.6 gFlops for SP comes from 3.2*4*2. Here 3.2 is the SPE clock speed, and in each cycle, it is possible to issue one multiply-add instruction (4 floating points vector, 8 operations). On the other hand, for double precision vectors, there is a six cycle issue latency, which means only one DP operation can be issued in 7 cycles. Using DP vector and multiply-add instruction, it is possible to get 3.2*2*2/7 = 1.83 gFlops. The interesting thing is if other operations are used (say add or multiplication) instead of multiply-add, the peak gFlops will be halved 0.91. Therefore, it is not possible to get more than 50% of peak gFlops, if multiply-add is not
used. In addition, sometimes data dependency, other operations, branch operations reduce the effective gFlops.

We do an experiment to get peak gFlops in Cell. Our code (which computes Harmonic number using multiply-add) achieves 1.78 gFlops for DP, and 22.37 gFlops for SP. The little bit reduction in peak gFlops occur due to some compulsory access to LS, and branch operations at the end of loop. The important thing to note is peak gFlops is only possible when multiply add operations are used, and it is possible to issue DP operations in every 7 cycles (or for SP operations in every cycle).

3. Parallelization of 3D Jacobi Method

The 3D Jacobi method is an iterative algorithm that updates a 3D grid multiple times. In each iteration, each point of the grid is updated using the following equation –

\[
Un(i, j, k) = \left( U(i-1, j, k) + U(i+1, j, k) + U(i, j-1, k) + U(i, j+1, k) + U(i, j, k-1) + U(i, j, k+1) \right) \times c
\]

Here \( U \) is the main grid array, and \( Un \) is another grid array that stores updated values of \( U \). We use \( N \) as the grid size, and \( I \) as the number of iterations. We denote \( x \)-axis to represent columns, \( y \)-axis to represent rows, and \( z \)-axis to represent planes/pages of the 3D grid.

The memory space of Jacobi method is large \( O(N^3) \) and the time complexity in serial case is \( O(IN^3) \). Because of accessing data in three dimensions, the Jacobi method is very memory intensive.

In Jacobi method, it is possible to update all grid points in parallel because the update only requires the values from previous iteration. Therefore, the simple way to parallelize Jacobi method is to divide the grid into \( K \) sub-grids (\( K \) is the number of threads or processes), and execute each sub-grid in one processor. However, there is data dependency between consecutive iterations, and sub-grids need the points from neighboring sub-grids to compute the value of its boundary points. Hence, all threads need to synchronize at the end of each iteration. A simple barrier mechanism can do that. The parallel implementation does not increase the space complexity, and ideally, linear speed up is possible (as there is no serial section in the parallel implementation). However, being memory intensive, the parallel implementation performance is somewhat limited by available memory bandwidth, or memory system efficiency.

Processor geometry is an important factor in traditional systems, as communication cost, cache utilization affect the performance a lot. As a result dividing the grid into sub-grids should consider the optimal processor geometry for a system.

4. Parallel Jacobi Method in Cell

To implement the parallel Jacobi method in Cell, we use the basic parallelization idea described above. PPE divides the grids into \( K \) (\( K \) is the number of SPEs) sub-grids, and feed each SPE a sub-grid. The SPEs process each sub-grid and synchronize at the end of each iteration. Each SPE computes the error for its sub-grid and writes to memory. PPE computes the final error from the set of errors provided by SPEs.

4.1. Vectorization

Jacobi method is easy to vectorize. We consider each two consecutive grid points along \( x \)-axis as a double precision vector (set of two double points). In this way, we can do the vector additions for \( y \)-axis, \( U(i, j-1, k)+U(i, j+1, k) \), and \( z \)-axis, \( U(i, j, k-1)+U(i, j, k+1) \) easily, but needs some further manipulation for vector additions along \( x \)-axis, \( U(i-1, j, k)+U(i+1, j, k) \). To explain it, let us assume four vectors – \((a0, b0), (a1, b1), (a2, b2), (a3, b3)\). Taking the addition of adjacent vectors, we have – \((a0+a1, b0+b1), (a1+a2, b1+b2), (a2+a3, b2+b3)\).
Now for vector \((a_1, b_1)\) we need \((b_0+b_1, a_1+a_2)\), which we can get using shuffle operation. Using this technique, for vector addition along x-axis requires one vector addition, and one shuffle operation. Therefore, equation (1) can be completely vectorized using five vector additions, one shuffle, and one multiplication.

However, in our implementation we use four vector additions, one shuffle, one multiplication, and one multiply-add operation. This gives more instruction level parallelism. To explain it, let us consider the vectors as \(v_1\) to \(v_6\), and \(c\). The following table shows the number of cycles required in each of the cases –

<table>
<thead>
<tr>
<th>Case 1 (5 add, 1 multiplication)</th>
<th>Case 2 (4 add, 1 multiplication, 1 multiply-add)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_1 = v_1 + v_2) cycle 1 to 13</td>
<td>(T_1 = v_1 + v_2) cycle 1 to 13</td>
</tr>
<tr>
<td>(T_2 = v_3 + v_4) cycle 7 to 19</td>
<td>(T_2 = v_3 + v_4) cycle 7 to 19</td>
</tr>
<tr>
<td>(T_3 = v_5 + v_6) cycle 13 to 25</td>
<td>(T_3 = v_5 + v_6) cycle 13 to 25</td>
</tr>
<tr>
<td>(T_4 = T_1 + T_2) cycle 20 to 32</td>
<td>(T_4 = T_1 + T_2) cycle 20 to 32</td>
</tr>
<tr>
<td>(T_5 = T_3 + T_4) cycle 33 to 45</td>
<td>(T_5 = T_3 \times c) cycle 26 to 38</td>
</tr>
<tr>
<td>(T_6 = T_5 \times c) cycle 46 to 58</td>
<td>(T_6 = T_4 \times c + T_5) cycle 39 to 51</td>
</tr>
</tbody>
</table>

Table 2: Number of cycles required for two different implementations.

From Table 2, Case-2 takes 7 cycles less than Case-1 despite using one more double precision floating point operations (7 vs. 6).

4.2. Dividing the grid into sub-grids

Workload distribution among SPEs or dividing the grid into sub-grids is an important part of the algorithm design. Because, the coding complexity, frequency of DMA operations, utilization of DMA transfer capacity depend on how the grid is divided into sub-grids. Three primary ways to cut the grid are – cut along x-axis, cut along y-axis, cut along z-axis. Figure 2 depicts the three scenarios –

![Figure 2: Dividing the grid into sub-grids in different ways](image)

In the above figure, it shows the three ways to cut the grid into 4 sub-grids (say we have 4 SPEs). If the grid is cut along x-axis (each row is divided into 4 sets of columns), then for each row segment, it requires a different DMA operation. To compute the points of a row, it also requires four other neighboring rows. In this way, computation for \(N\) points of a particular row requires 5 DMA operations.
for 1 SPE, and 20 DMA operations by four SPE. Cutting along y-axis (dividing a page into 4 sets of rows) allows one DMA operation to fetch multiple rows (as consecutive rows are adjacent in memory). However, it may face problem of not utilizing DMA transfer capacity fully. For example for $N=64$, each DMA operation can fetch at most $16 \times 64 \times 8 = 8K$ whereas max transfer size possible is 16K. This will increase the number of DMA operations eventually. Cutting along z-axis solves all problems, as in this case, each SPE gets a sub-grid, and the sub-grids do not interleave with one another in memory. Now it is possible to use full capacity DMA transfer. Other processor geometries like (cut both along y, z axes will make these issues more complicated and will not work well in Cell).

In our implementation, we cut the grids along z-axis, and divide planes evenly. For example, for $N=64$, each SPE gets 16 planes. If even distribution is not possible we try to make it as even as possible by giving one extra plane to some of SPEs. This type of load distribution approach is also scalable. We can use as many SPEs as we want until it crosses $N$. In addition, we can use this approach (with MPI) if multiple CBE are used.

### 4.3. DMA Operations

In each iteration, each SPE processes a set of planes of the main grid. During each DMA operation, the algorithm brings as many rows as it can fit in 16K (but not more than $N$ rows, i.e. not more than a plane). If $P$ rows fit in 16K, the algorithm first processes the first $P$ rows of all planes, then it processes the next $P$ rows of all planes, and the process continues in this way. This approach allows reuse of data from previous step. The following figure explains the technique.

![Figure 3: DMA Operations](image)

In the above figure, at most 28 rows can be packed in one DMA operations. So, the algorithm first performs DMA 1 to three, then it processes the computation for Plane 1. In next step, it does DMA 4, and performs computation for Plane 2. Thus, it reuses the data of two planes in each step. Once it computes the first set of rows for all planes, it starts with the second set of rows.

### 4.4 Synchronization

PPE uses mailboxes to synchronize SPEs at the end of current iteration. SPEs put values in their outbound mailboxes after completing current iteration, and they wait until they get some value in their inbound mailboxes from PPE. PPE once it receives values from all SPEs in their outbound mailboxes, put the “start again” signal in the inbound mailboxes of all SPEs. SPEs start their next iteration then. While SPEs are doing computation, PPE waits. This approach is similar to implement a barrier, where PPE works as a controller of the SPEs.

However, the above algorithm (despite using maximum data reuse, full capacity DMA operations, swapping grid $U$, $Un$ to eliminate copying) achieves poor gFlops, 0.25, which is around 20-25% for the possible peak gFlops for Jacobi method. Next section shows, how the performance of the algorithm can be dramatically improved.
5. Performance Optimizations

To improve the performance, we add three basic optimization techniques –

- Process multiple points at the same time
- Shuffling instructions
- Double buffering

The previous algorithm processes one point (or vector) at each step. This is not efficient as it requires six load operations per point. Also, data dependency creates unnecessary stalls (shown in Table-2). Processing multiple points (which are spatially adjacent) at the same time gives two main benefits – re-use of points, and providing more independent instructions.

It is important to note that, data sharing occurs in three dimensions. This implies that if multiple points are processed, they should come from different planes, rows, and columns. Processing 2x2x2 size small sub-grid is better than processing 4x2 size plane, or eight size rows or columns. Processing more points simultaneously gives more savings. In our implementation, one inner loop iteration processes 24 (3x4x2) points in two steps. The first step processes 12 points of first column from 3 consecutive planes (4 1st column points per plane), and the next step processes 12 points of 2nd column of those planes.

\[
\begin{array}{|c|c|c|}
\hline
\text{Column 1} & \text{Column 2} \\
\hline
\text{Row 1} & A111 & A112 \\
\text{Row 2} & A121 & A122 & \text{Plane 1} \\
\text{Row 3} & A131 & A132 \\
\text{Row 4} & A141 & A142 \\
\text{Row 1} & A211 & A212 \\
\text{Row 2} & A221 & A222 & \text{Plane 2} \\
\text{Row 3} & A231 & A232 \\
\text{Row 4} & A241 & A242 \\
\text{Row 1} & A311 & A312 \\
\text{Row 2} & A321 & A322 & \text{Plane 3} \\
\text{Row 3} & A331 & A332 \\
\text{Row 4} & A341 & A342 \\
\hline
\end{array}
\]

In this way, to process 24 points, it needs total 76 points, of which it is possible to re-use 24 points while computing for Column 3. Thus, effective load operations per point become 2.167. However, it is not always possible to process 24 points. We handle those cases by processing 18 (3x3x2) points, or 16 (2x4x2) points, or 12 (2x3x2) points, or eight (2x2x2) points. As a result, our implementation can extract good performance for various grid sizes. We only impose one restriction, grid size should be a multiple of four. This restriction reduces further coding complexity.

We also manually shuffle instructions to increase dual cycle issue. It gives little bit of performance improvement for our implementation. Double buffering is another technique that boosted the performance improvement. Double buffering hides the DMA operations time. In our implementation, we use double buffering for both load, and store operations. Double buffering really increases the effective gFlops significantly. We also use branch hints wherever applicable. The final algorithm is summarized in the following.

Algorithm

1. Use DMA to get grid points for 5 planes (one extra plane above 3 planes, one plane below 3 planes)
2. While doing the computation with 5 planes, load next 3 planes in another buffer, B’ [double buffering for loads]
3. After computation is done, store the output Un in memory using DMA
4. Use two planes from previous five planes, and three planes from $B'$. Do the computation using a different Un. [double buffering for stores]

5. Repeat steps 2 to 4, for each set of rows

6. Repeat steps 2 to 5, for each iteration

The above algorithm uses LS effectively. To keep planes for input data, it requires $(5+3) = 8$ planes, and for output data, it requires six planes. Each plane takes 16K space, thus in total 224K is used out of 256K.

6. Methodology

All the implementations use SDK 3.0, a C/C++ language extension for Cell Programming. We use the PS3 (dark.ucsd.edu) machine to develop the code, and make the final performance testing on the Georgia Tech IBM QS20 dual-Cell blade cluster, CellBuzz [3]. To annotate the SPU program’s assembly code with cycles taken by each instruction, the spu-timing utility is used. For performance comparison with other architectures, we use the NCSA Intel 64 Cluster, Abe [4]. Each processing node of Abe has eight cores, constructed from dual-socket, quad-core Intel 64 (Clovertown, E5345) 2.33 GHz processors. The L2 cache capacity per node is 2x4 MB, and front side bus is 1333 MHz (per socket). A detailed specification can be found in [4]. We compare a threaded version of 3D Jacobi method with the implementation in Cell using eight SPEs.

7. Results

In this section, we first present the theoretically peak gFlops possible for Jacobi method, then we present the results with/without double buffering, scaling, load balancing, comparison with Abe, and finally some static profiling info. We run all our simulations using “numactl –interleave=all”. In CellBuzz, a QS20 blade is a NUMA machine. Using “numactl” helps to exploit memory interleaving, and Rambus bandwidth well. Without using “numactl” the gFlops rate that we get is not consistent, and sometimes vary by more than 1 gFlops for 8/16 SPEs. Using “numactl”, we always get consistent timing. All the results reported here use “numactl”.

7.1. Peak gFlops

We use the 4 additions, 1 multiplication, and 1 multiply-add approach to compute the value of one double precision vector, as it is the optimal way to get peak performance (described in section 4.1). Using more multiply-add will give more gFlops, but the running time will not improve. With this approach, if one DP vector is updated in each loop iteration, minimum number of cycles is 51, and possible peak gFlops per SPE is $7*2*3.2/51 = 0.88$ gFlops. On the other hand using 24 DP vectors (assuming no dependency stall in this case) in one iteration, minimum number of cycles required is $24*6*7+6=1014$ cycles, giving $24*7*2*3.2/1014 = 1.06$ gFlops. It assumes no dependency stalls, but this is not the case as there are some compulsory loads in every iteration. Moreover, the static profiling (described later) shows that possible peak is around 0.96 gFlops.

In our experiment, we calculate two different gFlops –

- **Actual gFlops**: The SPE code computes it. In this case, only the time taken to execute each set of three planes (or two planes or one plane) is computed, and the time taken for DMA operations, function calling is ignored. SPU_READ_DECREMENTER is used to get accurate timing. However, this gFlops is not very useful as it does not correspond to actual running time correctly, and only shows the efficiency of the main program kernel.

- **Effective gFlops**: The PPE part of the program computes it. We take the time taken to run all iterations in this case. Therefore, this gFlops actually corresponds to actual running time.
The following table shows the actual gFlops (per SPE) for different number of SPEs, and different grid sizes.

<table>
<thead>
<tr>
<th>#of SPE</th>
<th>N</th>
<th>1 gFlops/SPE</th>
<th>2 gFlops/SPE</th>
<th>4 gFlops/SPE</th>
<th>8 gFlops/SPE</th>
<th>16 gFlops/SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64</td>
<td>0.945559</td>
<td>0.944325</td>
<td>0.942464</td>
<td>0.937632</td>
<td>0.930462</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.944765</td>
<td>0.944176</td>
<td>0.943024</td>
<td>0.940712</td>
<td>0.936249</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.936558</td>
<td>0.936287</td>
<td>0.935624</td>
<td>0.934574</td>
<td>0.932254</td>
</tr>
</tbody>
</table>

Table 3: Actual gFlops per SPE for different grid sizes, and number of SPEs

From Table-3, it is clear that main computation part is getting very close to (around 90%) theoretically possible peak gFlops. The slight variation we see occurs due to using various sets of points (24, 18, or 12 etc.) depending on different grid sizes, and each SPE’s share. As actual gFlops do not correspond to the runtime, we will only use effective gFlops in next results.

### 7.2. Scaling with/without Double Buffering

For memory intensive applications, double buffering is very important for getting good gFlops. Though DMA operations are very fast in Cell, without double buffering some useful cycles are wasted for waiting data. We make several experiments to see the effect of double buffering.

![Graphs showing gFlops with and without double buffering for different grid sizes and SPE numbers.](image)

Figure 4: gFlops w/o Double Buffering for various grid sizes (5000 iterations). (a) Without double buffering (b) With Double Buffering (c) Comparison for grid size 256.
So, from figure-4, it is clear that double buffering is crucial for memory intensive applications, and it can boost performance a lot by pre-fetching necessary data. In fact, double buffering does an excellent job to hide the memory access latency, which slows down the performance in traditional architectures. The graph also shows, the application scales well without double buffering. Nevertheless, with double buffering it does not scale well beyond eight SPEs. This is probably because of bandwidth limitation. We will discuss it in following sections. Another important thing is the performance drop for 16 SPEs (with double buffering) in case of grid size 64. In this case, each SPE gets only four planes, so double buffering cannot be used effectively. Also using small grid sizes, synchronizes cost is relatively higher (SPEs are done with their part quickly).

7.3. Load Balancing and Comparing with Abe

We compare the performance of Jacobi method in Cell BE with the threaded implementation in Abe (8 core node, shared memory). To get a broader view about load balancing, and other issues, we run the experiment with different grid sizes. All the simulations use 5000 iterations, and Cell implementation use double buffering.

![Figure 5: Performance comparison with Abe](image)

We use the best processor geometry in case of Abe, which is 1x4x2 in our implementation. As the grid sizes are multiple of four, threads in Abe are all load-balanced. However, for Cell BE implementation, in some cases SPEs are unbalanced, say using $N=244$, 12 SPEs get 15 planes each, and four SPEs get 16 planes each. The interesting thing is for the grid sizes that we test, load unbalancing by one plane does not cause too much overhead. In fact, sometimes, it gets better performance. This happens because, in such cases full capacity DMA operations occur more ($N=124$), or set of 24 vector computations occur more ($N=244$), etc. The relationship is pretty complex, and load balancing is not the only issue to extract more performance from Cell. The other interesting thing is both 8 SPE, and 16 SPE implementation follow the same trend. Both achieve peak performance for $N=244$, for 8 SPE it is 6.135 gFlops, and for 16 SPE it is 6.746 gFlops. In addition, both do badly for $N=260$.

Comparing with Abe, we see Abe performs badly once grid size gets large, and the sub-grid does not fit in local cache. On the other hand, Cell maintains very good gFlops for different grid sizes. This is
possible due to Cell’s memory access latency hiding using double buffering. For grid sizes larger than 124, Cell BE (eight SPE) gives 6-7x speedup over Abe.

7.4. Static Profiling
Using the spu_timing utility, and some calculation, we get the following information for one iteration of the inner loop of main computation kernel. Each iteration of the loop computes the value for 24 vectors. The outer loop contains many necessary operations, but the inner loop performance is the dominant one. That is why we only do the calculation for a single iteration of the inner loop.

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Cycles</th>
<th>Dual Cycle Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>299</td>
<td>Total</td>
</tr>
<tr>
<td>DP Vector</td>
<td>144</td>
<td>1121</td>
</tr>
<tr>
<td>LD</td>
<td>52</td>
<td>1014</td>
</tr>
<tr>
<td>ST</td>
<td>24</td>
<td>113</td>
</tr>
<tr>
<td>Shuffle</td>
<td>24</td>
<td>0.2997</td>
</tr>
<tr>
<td>Integer</td>
<td>39</td>
<td>0.9591</td>
</tr>
<tr>
<td>Others</td>
<td>16</td>
<td>42</td>
</tr>
<tr>
<td>Even pipe</td>
<td>183</td>
<td>Integer operations</td>
</tr>
<tr>
<td>Odd pipe</td>
<td>116</td>
<td>Others</td>
</tr>
</tbody>
</table>

Table 4: Static profile info for one iteration of main inner loop

From Table-4, we see our implementation uses 90.45% of its cycles for doing double precision operations for the crucial inner loop iteration. Only 9.55% cycles are used to do the data loading from LS, storing to LS, shuffle operations, integer additions, and others. Also, the implementation exploits dual issue very well, as all of integer operations are dual issued with either LD/ST operations, or shuffle operations. The 9.55% cycles are wasted because of data loading from local store, and the dependency of vector operations on that data. We see a peak gFlops of 0.9591 per iteration, which reduces to 0.93 to 0.945 (Table-3) because of outer loop operations, branch instructions at the end of each loop iteration, etc.

7.5. Bandwidth considerations
We use some simple calculations to measure the memory bandwidth requirement for our implementation. The following table shows the bandwidth calculation for both 8, and 16 SPE implementation using a grid size of 256, and 5000 iterations.

<table>
<thead>
<tr>
<th># of SPE</th>
<th># of Input planes per SPE</th>
<th># of out planes per SPE</th>
<th>Total data brought in LS (GB)</th>
<th>Total data written in memory (GB)</th>
<th>Total data transfer (GB)</th>
<th>Time (sec)</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>34</td>
<td>32</td>
<td>679.71</td>
<td>639.72</td>
<td>1319.43</td>
<td>99.19</td>
<td>13.30</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>16</td>
<td>719.69</td>
<td>639.72</td>
<td>1359.42</td>
<td>93.21</td>
<td>14.58</td>
</tr>
</tbody>
</table>

Table 5: Bandwidth calculation

Considering effective peak bandwidth 21 GB/s, 8 SPE implementation uses 63% of the total bandwidth. Bandwidth traffic is normally bursty (when all SPEs start next iteration bandwidth requirement is high,
during synchronization low bandwidth requirement, etc), and it shows that the bandwidth utilization is very high. This gives another indication that if 3D Jacobi method in Cell is implemented using single precision floating points, it will be bandwidth limited. SP operations are more than 10 times faster in Cell. But with using possible peak bandwidth 21 GB/s, the speed up will be at most \( 2 \times 21/13.30 = 3.16x \) over DP implementation.

8. Conclusion

In this project, we implement the 3D parallel Jacobi method in Cell. The main goals of this project were to exploit Cell architecture, and to extract high performance for 3D Jacobi method in Cell. Our implementation with double buffering uses Cell architecture very well (uses large register files, dual cycle issue, branch hinting, etc.). Each SPE achieves more than 90% of peak gFlops when time for DMA operations is not considered. That means with infinite LS, the effective bandwidth will touch \( 8 \times 0.93 = 7.44 \) gFlops for eight SPE.

When everything is considered (DMA, synchronization, PPE-SPE communication, etc.), the effective gFlops is 0.925 for one SPE, around 6 gFlops for 8 SPE (per SPE 0.75 gFlops), and 6.7 gFlops (per SPE 0.42 gFlops). The current implementation is also scalable – the algorithm can be easily adapted to an MPI implementation for multiple CBE. The static analysis shows that the implementation does almost perfect things. To get more performance, the only option is to process more points in one loop iteration and thus reducing load operations, loop backward branch operations, etc. We also show that Cell can extract 6-7x speedup over traditional parallel machines, if the application is implemented in the right way.

9. References


