Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine

Fabrizio Petrini,[1] Gordon Fossum,[2] Juan Fernández,[3]
Ana Lucia Varbanescu,[4] Mike Kistler[2] and Michael Perrone[5]

[1]Pacific Northwest National Laboratory
Richland, WA 99352 USA
fabrizio.petrini@pnl.gov

[2]IBM Austin Research Lab
Austin, TX 78758, USA
{fossum,mkistler}@us.ibm.com

[3]University of Murcia
30071 Murcia, Spain
juanf@um.es

[4]Delft University of Technology
2628CD Delft, The Netherlands
A.L.Varbanescu@tudelft.nl

[5]IBM TJ Watson Research Center
Yorktown Heights, NY 10598, USA
mpp@us.ibm.com

Abstract

The Cell Broadband Engine (BE) processor provides the potential to achieve an impressive level of performance for scientific applications. This level of performance can be reached by exploiting several dimensions of parallelism, such as thread-level parallelism using several Synergistic Processing Elements, data streaming parallelism, vector parallelism in the form of 128-bit SIMD operations, and pipeline parallelism by issuing multiple instructions in the same clock cycle. In our exploration to achieve the optimum level of performance for Sweep3D, we have enjoyed many pleasant surprises, such as a very high floating point performance, reaching 64% of the theoretical peak in double precision, and an overall performance speedup ranging from 4.5 times when compared with "heavy iron" processors, up to over 20 times with conventional processors.

The research described in this paper was conducted under the Laboratory Directed Research and Development Program for the Data Intensive Computing Initiative at Pacific Northwest National Laboratory, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy under Contract DEAC0576RL01830.

1 Introduction

Over the last decade high-performance computing has ridden the wave of commodity computing, building cluster-based parallel computers that were able to leverage the tremendous growth in processor performance fueled by the commercial world. As this pace slows down, processor designers are facing complex problems when increasing gate density, reducing power consumption and designing efficient memory hierarchies. The scientific and industrial communities are looking for alternative solutions that can keep up with the insatiable demand of computing cycles and yet have a sustainable market outside the scientific world.

A major trend in computer architecture is the implementation of highly integrated chips. This trend is driving the development of processors that can perform functions that are typically associated with entire systems. Rather than building *monolithic* processors, prohibitively expensive to develop, with high power consumption and limited return on the technological investment, it is much more effective to build *modular* processors with multiple cores. These multi-core system-on-a-chip processors can integrate several identical independent processing units on the same die, together with network interfaces, acceleration units and other specialized units.

Several design avenues have been explored both in academia, such as the Raw multiprocessor and TRIPS, and

in the industrial world, with notable examples being the AMD Opteron, IBM Power5, Sun Niagara, Intel Montecito and many others [13, 11, 12]. Common themes across these processors are power, parallelism and memory performance. Experiencing severely diminished returns, microprocessors designers are turning their attention to thread-level parallelism. Explicitly parallel techniques, where a user or a compiler express the available parallelism as a set of cooperating threads, offer a more efficient means of converting power into performance than do techniques that must discover the implicit –and often limited– instruction level parallelism hidden in a single thread. Montecito embodies thread-level parallelism with two cores executing two threads each. Niagara is more extreme with eight cores of four threads each. This increase in explicit parallelism is in large part driven by power concerns: as chips push the limits of semiconductor and manufacturing technology, power-efficient designs are becoming essential to deliver more performance.

These architectural trends have profound implications on the system software and the compilation process, which is somehow reminiscent of the RISC vs CISC debate of a few decades ago. The burden is shifted from the architecture, which is becoming simpler and more streamlined, to the software, that is now required to extract several forms of parallelism and directly coordinate a plethora of computational and communication activities across various levels of memories and functional units.

The Cell Broadband Engine (Cell BE) processor, jointly developed by IBM, Sony and Toshiba, is a new member of the IBM Power/PowerPC processor family. The initial target was the PlayStation 3 game console, but its capabilities also make it well suited for various other applications such as visualization and image and signal processing. The Cell BE is a heterogeneous chip with nine cores (one control processor coupled with eight lightweight independent processing units) capable of massive floating point processing, optimized for compute-intensive workloads and broadband, rich media applications.

The disruptive processing power of the Cell BE, that with a frequency of 3.2 GHz peaks at 204.8 Gflops/second in single precision floating point and 14.63 Gflops/second in double precision, and its healthy connection to the commodity market combined with a low power consumption have not passed unobserved in the scientific community. This unprecedented level of performance is achieved by combining thread level parallelism, supported by the processing units, with vector parallelism with variable granularity, ranging from 16 parallel single-byte to 2 double precision operations.

Together with the initial excitement of some early evaluations [15], the scientific community is expressing several concerns on some important aspects. More specifically there is a general interest in understanding

- what is the *actual* fraction of the peak performance that can be actually achieved by scientific applications,
- the complexity of developing new applications,
- the complexity of developing new parallelizing compilers and
- whether there is a clear migration path for existing legacy software, most notably the applications that have already been written using popular communication libraries such as MPI [14].

In this paper we provide some insight and some preliminary answers by studying in depth the parallelization process of a scientific application, the radiation transport code Sweep3D.

Sweep3D has many interesting properties that makes it amenable for this exercise. On the one hand, Sweep3D is a realistic application that is representative of a substantial fraction computing cycles executed on some of the most powerful supercomputers in the world. On the other hand, it is a compact application, publicly available,[1] which has been extensively studied in previous publications, allowing direct apple-to-apples comparisons [4].

The paper discusses a parallelization strategy that is based on the existing MPI one, thus allowing a straightforward migration path, that uses five distinct levels of parallelism to exploit the full potential of the Cell BE processor. The initial, coarse-level MPI implementation is incrementally enhanced with thread level parallelism across the synergistic processing units, data streaming parallelism to deal with the space limitations of the local stores of these units, vector parallelism to execute multiple floating point operations in parallel and pipeline parallelism to guarantee dual issue of instructions when possible.

The complexity and the performance impact of each of these parallelization steps and other optimizations are quantified and discussed across the paper. Together with some unexpected surprises, such as a very high floating point performance achieved by the sequential parts of Sweep3D, reaching 64% of peak, we also experienced unexpected hurdles in the data orchestration, processor synchronization and memory access algorithms. Our initial implementation provides a performance speedup ranging from 4.5 times, when compared to "heavy iron" processors such as the IBM Power5, specifically designed for the scientific workload, to over 20 times when compared to conventional processors.

In the final part of the paper we also discuss how architectural improvements, such a a fully pipelined double-precision floating point unit and a higher memory and communication bandwidth, can affect the overall application performance.

[1] http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/

We believe that our work provides a valuable contribution for application developers, by identifying a software path that can be followed by other scientific applications. We expect that many of these hand-crafted techniques will eventually migrate into parallelizing tools and compilers. Processor designers will also find insightful information to develop the new generation of streaming processors, that will likely target from the very beginning the computing needs of scientific applications.

In order to make this paper self-contained, Section 2 describes the relevant architectural features of the Cell BE processor that are essential to understand our parallelization techniques, including the communication and synchronization protocols, the vectorization capabilities and the structure of the internal pipeline of the Synergistic Processing Elements. Section 3 briefly introduces Sweep3D and shows the computation and communication patterns of its current MPI implementation. Section 4 provides the "global view" of our multi-dimensional approach to parallelization. In order to get a higher level of performance, we have also used an arsenal of optimizations, described in Section 5. Finally, we compare our results with the ones obtained on other processors in Section 6, and we provide some concluding remarks in Section 7.

2. The Cell BE Processor

The Cell BE is a heterogeneous, multi-core chip capable of massive floating point processing, optimized for compute-intensive workloads and broadband, rich media applications. The Cell BE is composed of one 64-bit Power Processor Element (PPE), 8 specialized co-processors called Synergistic Processing Elements (SPEs), a high-speed memory controller and a high-bandwidth bus interface, all integrated on a single chip. The PPE and SPEs communicate through an internal high-speed Element Interconnect Bus (EIB). The latest Cell processor, running at 3.2 GHz, has a theoretical peak performance of 204.8 Gflops (single precision) and 14.63 Gflops (double precision). The EIB supports a peak bandwidth of 204.8 Gigabytes/second for intra-chip data transfers among the PPE, the SPEs, and the memory and the I/O interface controllers. The memory interface controller (MIC) provides a peak bandwidth of 25.6 Gigabytes/second to main memory. The I/O controller provides peak bandwidths of 25 Gigabytes/second inbound and 35 Gigabytes/second outbound.

The PPE is the main processor of the Cell BE, and is responsible for running the operating system and coordinating the SPEs. It is a traditional 64-bit PowerPC (PPC) processor core with a VMX unit, 32 KB Level 1 instruction cache, 32 KB Level 1 data cache, and 512 KB Level 2 cache. The PPE is a dual issue, in-order execution design, 2-way SMT processor.

Each SPE consists of a Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC) which includes a DMA controller, a Memory Management Unit (MMU), a bus interface and an atomic unit for synchronization with other SPUs and the PPE. The SPU is a RISC-style processor with an instruction set and a microarchitecture that is designed for high-performance on streaming and data-intensive computation. The SPU includes a 256 KB local scratchpad memory to store both the instructions and data of an SPU program. The SPU cannot access main memory directly, but it has to issue DMA commands to the MFC to bring data into the LS or write results back to main memory. The SPU can continue program execution while the MFC independently performs these DMA transactions. No hardware data load prediction structures exist for LS management, and each LS must be managed by software.

Each SPU has 128 128-bit SIMD registers. The large number of registers facilitates very efficient instruction scheduling and enables important optimization techniques such as loop unrolling. All SPU instructions are inherently SIMD operations that can run at four different granularities: 16-way 8-bit integers, 8-way 16-bit integers, 4-way 32-bit integers or single-precision floating-point numbers, or 2 64-bit double-precision floating point numbers.

The SPU is an in-order processor with 2 instruction pipelines. The floating point and fixed point units are on the even pipeline while the rest of the functional units are on the odd pipeline. Each SPU can issue and complete up to two instructions per cycle - one per pipeline. For a wide variety of applications, the SPU can approach this theoretical limit. All single-precision operations (8-bit, 16-bit, or 32-bit integers/floats) are fully pipelined and can be issued at the full SPU clock rate (e.g., four 32-bit floating point operations per SPU clock). The 2-way double precision floating point is partially pipelined, so its instructions issue at a lower rate (i.e., two double-precision flops every seven SPU clocks). When using single-precision floating-point fused multiply-add instructions (which count as two operations), the eight SPEs perform a total of 64 operations per cycle.

Communication mechanisms. To take advantage of all the computational power available on the Cell BE processor, work and data must be distributed and coordinated across the PPE and the SPEs. Each SPE has a DMA controller that performs high bandwidth transfers between the local store and main memory and between local stores. An SPE can also use either *signals* or *mailboxes* for short, low-latency (but also low-bandwidth) communication with other SPEs or with the PPE. More complex synchronization mechanisms are supported by a set of atomic operations available to the SPU that operate in a very similar manner to the `lwarx`/`stwcx` atomic instructions of the PowerPC architecture. In fact, the SPEs' atomic operations can seamlessly in-

teroperate with PPE's atomic instructions. Finally, the Cell BE allows memory-mapped access to nearly all resources on the SPEs, including the entire local store. This provides a convenient and consistent mechanism for special communications needs that are not met by the other techniques.

DMA Transfers. The MFC performs DMA operations to transfer data between the LS and system memory. DMA transfers are specified by an SPU program using fully compliant PPC virtual addresses. DMA operations can transfer data between the local store and any resources connected via the on-chip interconnect (i.e. main memory, the LS of another SPE, or an I/O device). SPE to SPE transfers can be sustained at a rate of 16 bytes (read) plus 16 bytes (write) every 16 SPU clock cycles, but at a much lower rate to main memory (i.e. aggregate main memory bandwidth is 25.6 GByte/sec for the entire Cell BE processor). The MFC accepts and processes DMA commands which are issued using the SPU channel interface or MMIO registers. DMA commands are queued in the MFC, and the SPU or PPE (whichever issued the command) can continue execution in parallel with the data transfer, using either polling or blocking interfaces to determine when the transfer is complete. This autonomous execution of MFC DMA commands allows DMA transfers to be conveniently scheduled to hide memory latency. The MFC supports naturally aligned transfers of 1, 2, 4, or 8 bytes, or a multiple of 16-bytes up to a maximum of 16 KB. DMA list commands can request a list of up to 2,048 DMA transfers using a single MFC DMA command. However, only the MFC's associated SPU can issue DMA list commands. A DMA-list is stored in the LS as an array of DMA source/destination addresses and lengths. When issued, the MFC is passed the address and length of the DMA-list in LS [6]. Peak performance can be achieved for transfers when both the EA and LSA are 128-byte aligned and the size of the transfer is an even multiple of 128 bytes.

3. Sweep3D

Sweep3D [10] solves a three-dimensional neutron transport problem from a scattering source. In general, "particle transport" (or "radiation transport") analyzes the flux of photons and/or other particles through a space. Its main usage is to facilitate the analysis of fires, explosions and even nuclear reactions without having to run experiments. For the discrete analysis, the space is divided into a finite mesh of cells and the particles are flowing only along a finite number of beams that cross at fixed angles. The particles flowing along these beams occupy fixed energy levels. The analysis computes the evolution of the flux of particles over time, by computing the current state of a cell in a time-step as a function of its state and the states of its neighbors in the previous time-step. For this implementation, the three-dimensional geometry is represented by a logically rectangular grid of cells (with dimensions I, J and K) divided into eight octants by the scattering source. The movement is modeled in terms of six angles (three angles in the forward direction and three angles in the backward direction) for each octant. The equations for each angle can be seen as a wavefront sweeping from one corner of the space (i.e., the octant) to the opposite corner. A complete sweep from a corner to its opposite corner is an iteration. There are several iterations for each time step, until the solution converges. Hence, the solution involves two steps: the streaming operator (i.e., result propagation), solved by sweeps, and the scattering operator, solved iteratively [1].

An S_n sweep for a single octant and a given angle works as follows. Each grid cell has 4 equations with 7 unknowns (6 faces plus 1 central). Boundary conditions complete the system of equations. The solution is reached by a direct ordered solver, i.e., a sweep. Three known inflows allow the cell center and three outflows to be solved. Each cell's solution then provides inflows to 3 adjoining cells (1 each in the I, J, and K dimensions). This represents a wavefront evaluation with a recursion dependence in all three grid directions. Each octant has a different sweep direction through the grid of cells, but all angles in a given octant are independent and sweep in the same way.

Sweep3D exploits parallelism via a wavefront algorithm [3, 5]. Each grid cell can be only computed when all the previous cells in the sweep direction have been already processed. Grid cells are evenly distributed across a two-dimensional array of processes. In this way, each process owns a three-dimensional tile of cells. The data mapping and the wave propagation during a north-to-south, west-to-east sweep is described in Figure 1. For the sake of simplicity, the K dimension is hidden for all processes other than upper-left corner process. The wave is originated by the process in the upper-left corner, that is, $P_{0,0}$. This process solves the unknowns of the local cells and then propagates the results to its east and south neighbors, that is, $P_{1,0}$ and $P_{0,1}$, respectively. At this point the two adjacent processes can start computing the first wave, while the upper-left corner process starts the second wave. In an ideal system, where computation and communication are perfectly balanced, each diagonal of processes would be computing the same wave at any given time.

The pseudo-code of the `sweep()` subroutine, the computational core of Sweep3D, is listed in Figure 2. Before each inner iteration (lines 7 to 17), the process issues waits for the I-inflows and J-inflows coming from the west and north neighbors, respectively (lines 5 and 6). Then, it computes the incoming wave through its own tile of cells using a stride-1 line-recursion in the I-direction as the innermost work unit (lines 8 to 15). It is worth noting that this ac-

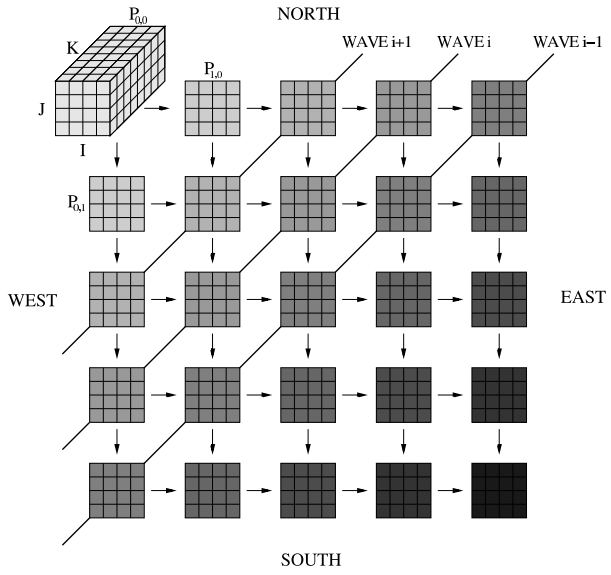

Figure 1. Data mapping and communication pattern of Sweep3D's wavefront algorithm.

cess to the arrays in a sequential fashion, can be exploited to extract further pipeline and data parallelism. Finally, the process sends the I-outflows and J-outflows to the east and south neighbors, respectively (lines 18 and 19).

```
1  SUBROUTINE sweep()
2  DO iq=1,8                          ! Octant loop
3   DO m=1,6/mmi                      ! Angle pipelining loop
4    DO k=1,kt/mk                     ! K−plane pipelining loop
5     RECV W/E                        ! Receive west/east I−inflows
6     RECV N/S                        ! Receive north/south J−inflows
7     DO jkm=1,jt+mk−1+mmi−1          ! JK−diagonals with MMI pipelining
8      DO il=1,ndiag                  ! I−lines on this  diagonal
9       IF .NOT. do_fixups
10       DO i=1,it                    ! Solve Sn equation
11       ENDDO
12      ELSE
13       DO i=1,it                    ! Solve Sn equation with fixups
14       ENDDO
15      ENDIF
16     ENDDO                          ! I−lines on this  diagonal
17    ENDDO                           ! JK−diagonals with MMI pipelining
18    SEND W/E                        ! Send west/east I−outflows
19    SEND N/S                        ! Send north/south J−outflows
20   ENDDO                            ! K−plane pipelining loop
21  ENDDO                             ! Angle pipelining loop
22 ENDDO                              ! Octant loop
```

Figure 2. The `sweep()` subroutine.

With this form of the wavefront algorithm, parallelism is limited by the number of simultaneous waves. In the configuration depicted in Figure 1, there would always be idle processes if $\sqrt{P}$ is greater than the number of simultaneous waves, an artificial limitation if Sweep3D is executed on a large configuration. To alleviate this problem, `sweep()` is coded to pipeline blocks of MK K-planes (MK must factor

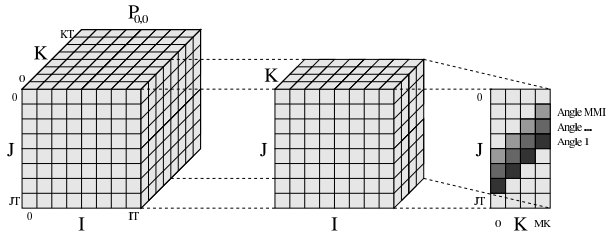

Figure 3. Sweep3D parallelization through K-plane and angle pipelining.

KT) and MMI angles (1 or 3) through this two-dimensional process array for each octant. As an example, the first inner iteration for $P_{0,0}$ is shown in Figure 3. This iteration processes a block of four K-planes (MK is 4) and eight J-planes (JT is 8) for three different angles (MMI is 3). The depicted `jkm` value is 6 which includes the sixth JK diagonal for angle 1, the fifth diagonal for angle 2 and the fourth diagonal for angle 3, that is, `il` is 12. One important feature of this scheme is that all the I-lines for each `jkm` value can be processed in parallel, without any data dependency. This property plays a central role in our proposed parallelization strategy discussed in the next section.

4. Parallelization Strategies

The shift of paradigm in architectural design imposed by the new multi-core processors is expected to determine a comparable, if not bigger, advancement in parallelizing compilers and run-time systems. However, even if early results [2] show that parallelizing compilers can achieve very good results on many significant application kernels, the efficient and automatic parallelization of full scientific applications is not yet feasible with state-of-the-art technology [7].

One straightforward approach to parallelize Sweep3D on a multi-core processor is by extending the wavefront algorithm to the SPEs [8]. The logical grid of processors can be simply refined by the presence of a larger number of computational units. Such a strategy may be difficult to integrate into a parallelizing compiler, because it requires detailed information about the process-level parallelization. More important, this strategy is not able to capture the multiple levels of parallelism of the Cell BE processor, which are essential to achieve high performance. Moreover, this solution does not address the data orchestration required to tackle the limited local storage available on the SPEs.

Our approach, graphically outlined in Figure 4 exploits five levels of parallelism and data streaming in the following way:

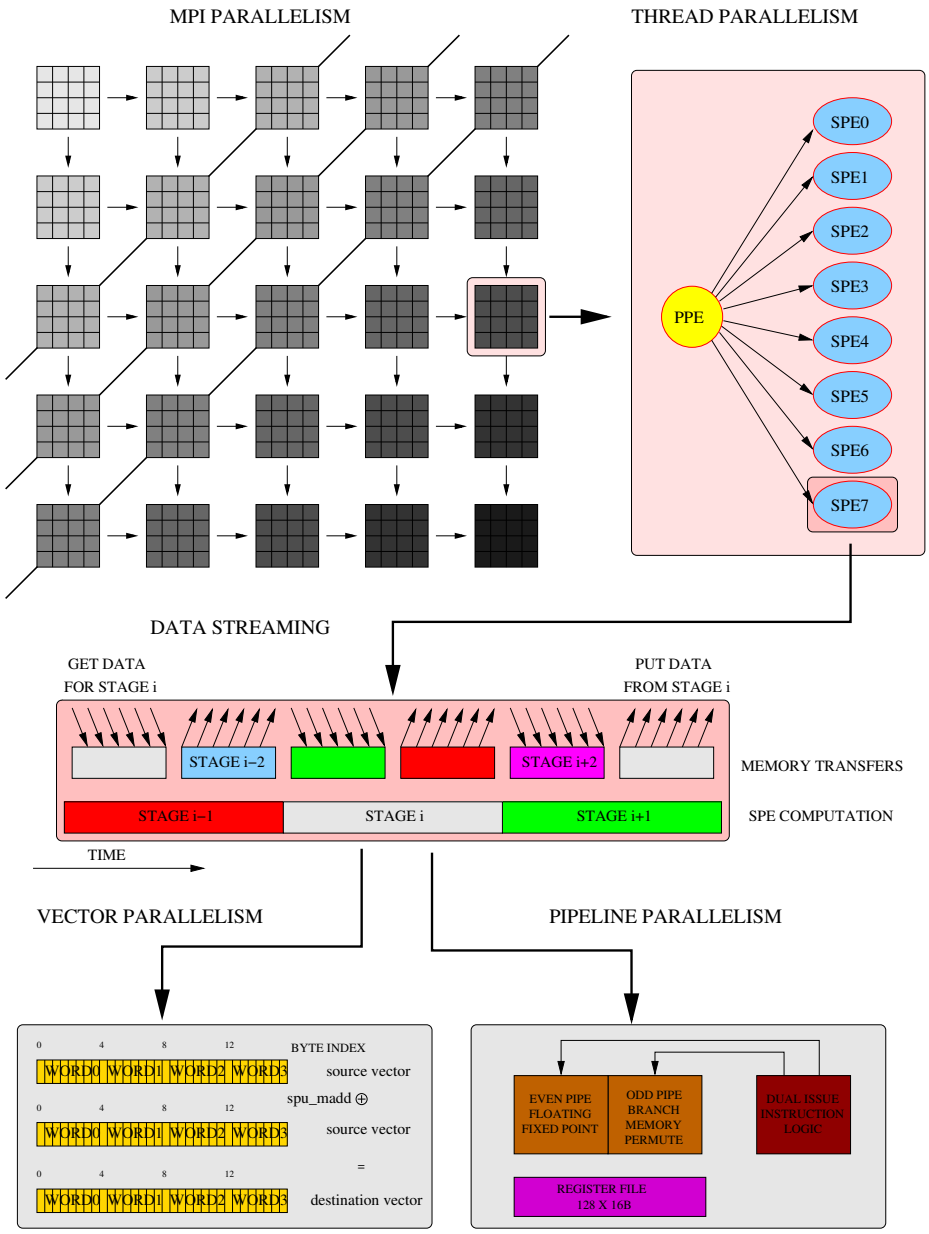

Figure 4. Parallelization process.

1. **Process-level parallelism**. At the highest level, we maintain the wavefront parallelism already implemented in MPI and other messaging layers; this guarantees portability of existing parallel software, that can be directly compiled and executed on the PPE without major changes (no use of the SPEs, yet).

2. **Thread-level parallelism**. We extract thread-level parallelism from the subroutine `sweep()` (lines 7 to 17), taking advantage of the lack of data dependencies for each iteration of the `jkm` loop. In our initial implementation, the I-lines for each `jkm` iteration are assigned to each SPE in a cyclic manner.

3. **Data-streaming parallelism**. Each thread loads the "working set" of the iteration using a double buffering strategy: before executing loop i, the SPE issues a collection of DMA gets to load the working set of loop $i + 1$ and DMA puts to store the data structures modified by loop $i - 1$.

4. **Vector parallelism**. Chunks of loops are vectorized in

groups of 2 words (double precision) or 4 words (single precision).

5. **Pipeline parallelism**. Given the dual-pipeline architecture of the SPEs, the application can use multiple logical threads of vectorization, increasing pipeline utilization and masking eventual stalls. Our double precision implementation uses four different logical threads of vectorization.

It is worth noting that our loop parallelization strategy does not require any knowledge of the global, process-level parallelization. And, given that it operates at a very fine computational granularity, it can efficiently support various degrees of communication pipelining.

5. Performance Optimization

On top of the layered parallelization, we have also used several optimization techniques to fine tune the performance of Sweep3D. In this section, we briefly explain these techniques, and we evaluate them in terms of their impact on the overall Sweep3D performance.

As a starting point for the experiments, we have ported Sweep3D to a platform containing a single Cell BE chip, running at 3.2 GHz. Sweep3D ran on the PPU alone with a 50×50×50 input set (50-cubed), without any code changes, in 22.3 seconds.

To prepare for efficient execution on SPUs, several additional steps were required:

1. all the arrays were converted to be zero-based,
2. multi-dimensional arrays were flattened (by computing the indices explicitly),
3. cache-line (128 bytes) alignment was enforced for the start addresses of each chunk of memory to be loaded into the SPU at run-time, to improve DMA performance [9]
4. the SPU code candidates (i.e., the code segments which took up most computation time) were identified,
5. a `memset` call was issued to zero out each big array.

To benefit from more aggressive optimizations, we have replaced Gnu's GCC compiler with the IBM XLC compiler, a mature, production quality compiler with excellent optimization support. After these changes, the execution time of the code (still running only on the PPE) was 19.9 seconds.

Next, we identified the segment of code to be executed on the SPEs. The loop structure was remodeled to split the computation across eight independent loop iterations. Now running on eight SPEs, the application execution time dropped to 3.55 seconds. This significant improvement is due to a combination of two factors: the speedup of the SPE versus the PPE, together with the parallelization across

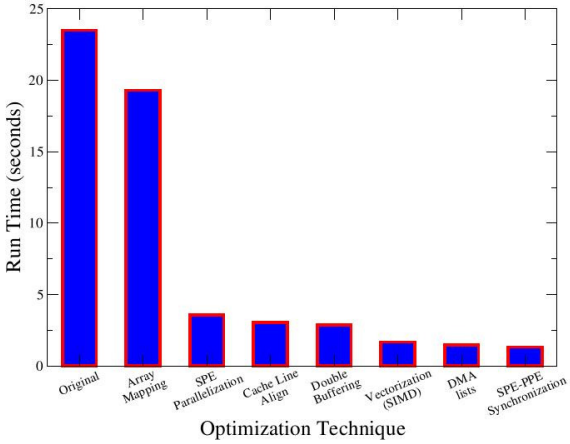

Figure 5. Performance impact of various optimizations.

8 cores. Similar drops in execution times are to be expected for any application that is able to fully utilize the eight SPE cores of the Cell BE.

After modifying the inner loop to eliminate goto statements and the array allocation to ensure that the rows of the "multi-dimensional" arrays are 128-byte aligned, the run time decreased to 3.03 seconds. Double buffering reduced the execution time to 2.88 seconds.

By using explicit SPE intrinsics (manual SIMDization, a technique common to many vector machines), we have achieved a new significant improvement, bringing the run time down to 1.68 seconds. A code snippet illustrating the SIMDization process is provided in Figures 6 (before) and 7 (after).

Finally, converting the individual DMA commands to DMA lists, and adding offsets to the array allocation to more fairly spread the memory accesses across the 16 main memory banks, further reduced the execution time to 1.48 seconds. Eliminating the use of mailboxes, and using a combination of DMAs and direct local store memory poking from the PPE to implement a PPE-SPE synchronization protocol, the execution time dropped to 1.33 seconds. Figure 5 summarizes the performance impact of all the optimizations.

5.1. Loop Vectorization

A closer look to the performance numbers presented in the previous paragraphs reveals the key role that vectorization, together with the SPE thread parallelism and the data orchestration, are playing in reducing the execution time of Sweep3D. Among the three, vectorization has the biggest impact in terms of relative gain. Thus, we have included a short example showing a snippet of code 'before' and 'after' the vectorization - see Figures 6 and 7, respectively.

```
for( n = 1; n < nm; n++ )
  for( i = 0; i < it; i++ )
    Flux[n][k][j][i] = Flux[n][k][j][i] + pn[iq][n][m]*w[m]*Phi[i];
```

Figure 6. Scalar loops

```
for( n = 1; n < nm; n++)
{
  vector double pnvalA, pnvalB, pnvalC, pnvalD;
  pnvalA = spu_splats(LS_pn[0][comp][n]);
  pnvalB = spu_splats(LS_pn[1][comp][n]);
  pnvalC = spu_splats(LS_pn[2][comp][n]);
  pnvalD = spu_splats(LS_pn[3][comp][n]);

  pnvalA = spu_mul(pnvalA, wVA);
  pnvalB = spu_mul(pnvalB, wVB);
  pnvalC = spu_mul(pnvalC, wVC);
  pnvalD = spu_mul(pnvalD, wVD);

  FluxVA = (vector double *) LS_Flux[0][comp][n];
  FluxVB = (vector double *) LS_Flux[1][comp][n];
  FluxVC = (vector double *) LS_Flux[2][comp][n];
  FluxVD = (vector double *) LS_Flux[3][comp][n];

  for( i = 0; i < (it +1)>>1; i++ )
  {
    FluxVA[i] = spu_madd(pnvalA, PhiVA[i], FluxVA[i]);
    FluxVB[i] = spu_madd(pnvalB, PhiVB[i], FluxVB[i]);
    FluxVC[i] = spu_madd(pnvalC, PhiVC[i], FluxVC[i]);
    FluxVD[i] = spu_madd(pnvalD, PhiVD[i], FluxVD[i]);
  }
}
```

Figure 7. Vectorized loops.

The following are a few hints to understand the transformation of the scalar loops into vectorized loops, and, consequently, the way the vectorization has been performed in the case of Sweep3D:

1. The SIMDized version performs operations on four separate threads of data simultaneously (A, B, C and D).

2. `wmVA`, `wmVB`, `wmVC`, `wmVD` represent a vectorized portion of `w[m]`. In this case, the values stored in the four vectors are interleaved because of the four logic threads.

3. The number of elements in any of the `wmV*` vectors is $(16/sizeof(w[m]))$, which is 2 in this case because `w[m]` is a double precision floating point value.

4. `spu_madd` does a double-precision 2-way SIMD `multiply_add`.

5. `spu_splats` replicates a value across a vector variable.

6. the inner loop runs half as many times because each operation is 2-way SIMD.

While the natural choice for vectorizing code is innermost loop unrolling, it requires this loop iterations to be data-independent. The innermost loop of Sweep3D processes one sweep, and iterations are data dependent, while the outer loop deals with time-step iterations, which are data-independent. Thus, we have chosen to vectorize along the time-wise loop, using four simultaneous logical threads, a solution that allows avoiding the data dependency stalls.

Peak and actual floating point performance. The Cell Broadband Engine can fully pipeline single-precision instructions (assuming dependency stalls are avoided), meaning that it is possible to compute a single-precision vector result (eight Flops, if it is an `spu_madd` instruction) every cycle. However, the current implementation of the Cell BE does not permit full pipelining of double-precision results. The hardware can handle only one double-precision vector math operation (two-way SIMD, each executing two operations such as a multiply_add) every seven cycles. On a 3.2 GHz system, this is equivalent to 14.63 Gflops/second.

The vectorized version of loop listed in Figure 8 (for the sake of simplicity and readability shown in scalar form) takes 590 cycles ("do_fixup" off) and 1690 cycles ("do_fixup" on) to execute 216 Flops. There are 24 and 85 instances of dual issue, respectively. This means that roughly 5% of the cycles are successfully issuing two commands per cycle. Thus, the theoretical peak performance is 4 Flops every 7 cycles, which is equivalent to 64% of the theoretical peak performance in the "do_fixup off" case.

```
for( i = i0; (i0 == 1 && i <= i1) || (i0 == it  && i >= i1) ; i=i+i2 )
{
  ci = mu[m]*hi[i];
  dl = ( Sigt[k][j][i] + ci + cj + ck );
  dl = 1.0 / dl;
  ql = ( Phi[i] + ci*Phiir + cj*Phijb[mi][lk][i] + ck*Phikb[mi][j][i]);
  Phi[i] = ql * dl;
  Phiir          = 2.0*Phi[i] - Phiir;
  Phii[i]        = Phiir;
  Phijb[mi][lk][i] = 2.0*Phi[i] - Phijb[mi][lk][i];
  Phikb[mi][j][i]  = 2.0*Phi[i] - Phikb[mi][j][i];
}
```

Figure 8. Computational kernel.

In single precision, the number of Flops jumps to 432, and the number of cycles drops to approximately 200, but the theoretical maximum is now 8 Flops/cycle, so our efficiency reaches a still-respectable 25%.

6. Performance Comparison and Future Optimizations

In the previous section we have discussed in detail the optimization process by focusing on a specific input set with a logical IJK grid of 50x50x50 cells. Figure 9 shows the grind time, the normalized processing time per cell, as a function of the input size. We assume the input domain is

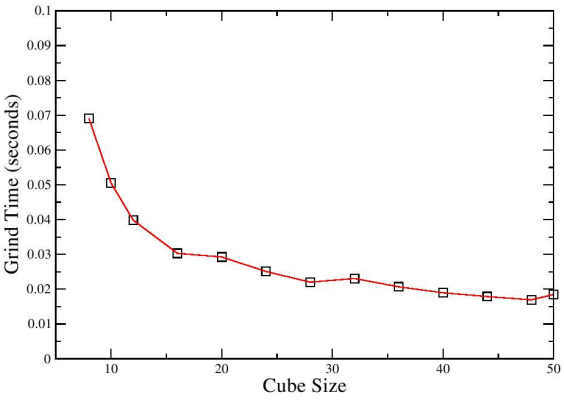

Figure 9. Grind time as a function of the cube size.

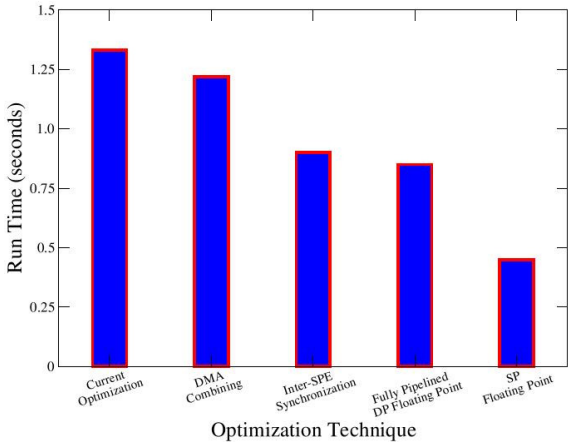

Figure 10. Expected performance impact of optimizations, architectural improvements and single precision floating point.

a three-dimensional cube of the specified size. For a cube size larger than 25 cells, the grind time is almost constant, indicating that our parallelization and performance analysis are robust with respect to the input size. Our load balancing algorithm farms chunks of four iterations to each SPE, so optimal load balancing can be achieved when the total number of iterations is an integer multiple of 4×8, as witnessed by the minor dents in Figure 9.

With a 50-cubed input size, the SPEs transfer 17.6 Gbytes of data. Considering that the peak memory bandwidth is 25.6 Gbytes/second, this sets a lower bound of 0.7 seconds to the execution time of Sweep3D. By profiling the amount of computation performed by the SPUs we obtain a similar lower bound, 0.68 seconds.

The gap between this bound and the actual run-time of 1.3 seconds, is mostly caused by the communication and synchronization protocols. We have identified the following directions to further reduce the execution time. The performance impact of these planned optimizations is outlined in Figure 10.

- By increasing the communication granularity of the DMA operations, which are currently implemented with lists of 512-byte DMAs (both for `puts` and `gets`), we can further reduce the run time to 1.2 seconds (as predicted by a test implementation, not yet fully integrated in Sweep3D).
- We noticed that the PPE cannot distribute efficiently the chunks of iterations across the SPEs, becoming a bottleneck. By replacing the centralized task distribution algorithm with a distributed algorithm across the SPEs, we expect to reduce the run time to 0.9 seconds.
- Contrary to our expectations, a fully pipelined double precision floating point unit would provide only a marginal improvement, to 0.85 seconds.

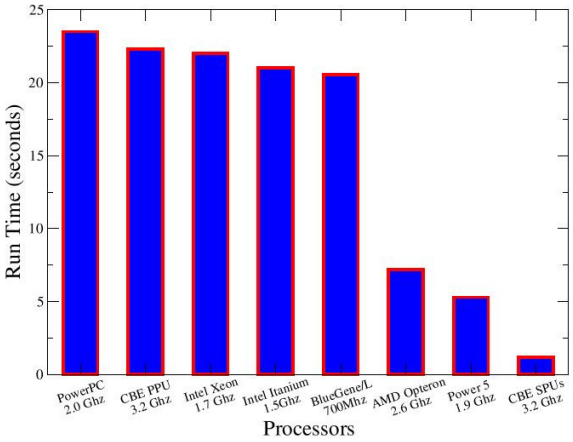

Figure 11. Performance comparison with other processors.

- By using single precision floating point, we expect a factor of 2 improvement, with a run time of approximately 0.45 seconds, again determined by the main memory bandwidth.

Figure 11 compares the absolute run time with other processors. The Cell BE is approximately 4.5 and 5.5 times faster than the Power5 and AMD Opteron, respectively. We expect to improve these values to 6.5 and 8.5 times with the optimizations of the data transfer and synchronization protocols. When compared to the other processors in the same figure, Cell BE is about 20 times faster.

7. Conclusions

Together with an unprecedented level of performance, multi-core processors are also bringing an unprecedented level of complexity in software development. We see a clear shift of paradigm from classical parallel computing, where parallelism is typically expressed in a single dimension (i.e, local vs remote communication, or scalar vs vector code), to the complex, multi-dimensional parallelization space of multi-core processors, where multiple levels of parallelism *must* be exploited in order to gain the expected performance. Furthermore, the multi-core design space complexity provides various opportunities to achieve, in a single chip, performance typical of entire clusters.

Probably the most important contribution of this paper, and also a fundamental lesson to be learned from it, is the exposure of this unavoidable multi-core complexity in a clear, unified manner, focusing on the multi-layered parallelization opportunities and challenges, and showing a possible, altough non-trivial path to follow for extracting the expected high level of overall performance.

A second important contribution is the proof that porting and optimizing a real life application, like Sweep3D, on a platform like Cell is worth the effort. Our initial Sweep3D parallelization is very fast, ranging from 4.5 times (with potential optimizations up to 6.5 in the near future) faster when compared with the IBM Power5, arguably the fastest "heavy iron" processor specifically designed for the scientific workload, to a factor of 20 on other processors.

Also, this paper has proven that the Cell BE processor, although initially designed mainly for computer games and animation, is a very promising architecture for scientific workloads as well. Even though the simple architectural design of its SPUs forces the user to consider a number of low-level details, it offers good opportunities to achieve high floating point rates: in our case we were able to reach an impressive 64% of peak performance in double precision (9.3 Gflops/second) and almost 25% in single precision (equivalent to 50 Gflops/second). We have also shown that the memory performance and the data communication patterns play a central role in Sweep3D, being currently the major bottleneck for this application. Most likely, other scientific applications will behave similarly.

References

[1] R. S. Baker, C. Asano, and D. Shirley. Implementation of the first-order form of the 3-D discrete ordinates equations on a T3D. Technical report, Los Alamos National Laboratory, Los Alamos, New Mexico, 1995.

[2] A. E. Eichenberger, K. O'Brien, K. K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the Cell Processor. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, Saint Louis, MO, August 2005.

[3] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications. *International Journal on High Performance Computing Applications*, 14(4):330–346, 2000.

[4] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *2000 International Conference on Parallel Processing, (ICPP'00)*, Toronto, Canada, August 2000.

[5] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of International Conference on Parallel Processing (ICPP'00)*, Toronto, Canada, Aug. 2000.

[6] IBM. *Cell Broadband Engine Architecture V1.0*. www-128.ibm.com/developerworks/power/cell/downloads_-doc.html.

[7] C. S. Ierotheou, S. P. Johnson, P. F. Leggett, M. Cross, E. W. Evans, H. Jin, M. A. Frumkin, and J. Yan. The Automatic Parallelization of Scientific Application codes using a Computer Aided Parallelization Toolkit. *Scientific Programming*, 9(2-3):163–173, 2001.

[8] D. Kerbyson and A. Hoisie. Analysis of Wavefront Algorithms on Large-scale Two-level Heterogeneous Processing Systems. In *Workshop on Unique Chips and Systems (UCAS2)*, Austin, TX, March 2006.

[9] M. Kistler, M. Perrone, and F. Petrini. Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*, 25(3), May/June 2006.

[10] K. Koch, R. Baker, and R. Alcouffe. Solution of the First-Order Form of Three-Dimensional Discrete Ordinates Equations on a Massively Parallel Machine. *Transactions of American Nuclear Society*, 65:198–199, 1992.

[11] P. Kongetira, K. Aingaran, and K. Olokotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, March/April 2006.

[12] R. Kota and R. Oehler. Horus: Large-Scale Symmetric Multiprocessing for Opteron Systems. *IEEE Micro*, 25(2):30–40, March/April 2006.

[13] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, March/April 2006.

[14] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming th Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, January 2006.

[15] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The Potential of the Cell Processor for Scientific Computing. In *ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2006.