

Experiments in scientific computation on the PlayStation 3

Erik O. D. Sevre · Monica D. Christiansen ·
Matt Broten · Shuo M. Wang · David A. Yuen

Received: 22 May 2008 / Revised: 26 May 2008 / Accepted: 26 May 2008 / Published online: 28 June 2008
© Springer-Verlag 2008

Abstract The Sony PlayStation 3 (PS3) offers the computational power of a parallel processor at low cost, which makes it a great starter unit for development in parallel programming. To explore the capabilities of the unit, we took a simple ray tracing program and extended it to render triangulated height field data across the PS3's 6 synergistic processing units (SPUs). We also implemented the heat averaging equation as a precursor to CFD analysis on the PS3. In our studies, we found the Cell engine in the PS3 to be a powerful machine, however great care must be taken while developing because its unique platform calls for many levels of optimization to ensure efficiency.

Introduction

In the past 2 years, there has been a lot of discussion about the new Cell Broadband Engine (Cell BE) microprocessor. The simple reason for this is that parallelized architectures, such as the Cell, are the future of scientific computation. This is because they enable a higher volume of calculations to be performed.

However, due to the cost of building a Cell computer, simply constructing a complete system for use as a test bed is impractical. The PS3, on the other hand, provides an

ideal environment to start developing with its native support for the Yellow Dog Linux operating system. Costing only \$600, the PS3 Cell provides 218 Gflops of processing power that can be utilized with a relatively quick and simple installation.

We wanted to see if the Cell processor lived up to the advertised capabilities. However, we needed to consider the limitations of the PS3 Cell: it has very limited memory, it has no access to the graphics processor unit (GPU), and it performs better using single precision floating-point operations. In this paper, we will discuss the applications we have written for the PS3 Cell processor, the many challenges we found when programming, and the solutions we came up with to overcome these challenges.

Many of the programs we look at on the PS3 can be accomplished using a GPU, but we have chosen to focus our attention on investigating how the Cell BE works as a computation engine. The Cell BE is midway between a conventional CPU and a GPU. Each processor has advantages and disadvantages based on how memory and computations are managed. GPUs excel at graphics, CPUs are good for general computing, and the Cell BE has been created for core scientific computing. Instead of comparing the PS3 to a GPU we instead look at how we developed and adapted code for the PS3 and looking at the problems and advantages we saw with the Cell BE.

Hardware description of the PS3

The PS3 features the novel multi-core Cell BE processor. The Cell BE processor is a heterogeneous chip comprised of nine cores. One core is a general purpose 64-bit 3.2 GHz PowerPC processor referred to as the power processing element (PPE). The other eight cores are 3.2 GHz single

For insight on early work done on modern stream computing and GPU processing look up the Merrimac project and Pat Hanrahan.

E. O. D. Sevre (✉) · M. D. Christiansen · M. Broten ·
S. M. Wang · D. A. Yuen
Department of Geology and Geophysics and Minnesota
Supercomputing Institute, University of Minnesota,
Minneapolis, MN 55455, USA
e-mail: esevre@i7.msi.umn.edu

instruction multiple data (SIMD) processors referred to as synergistic processing elements (SPEs) (Fig. 1).

Each SPE is comprised of a synergistic processing unit (SPU), a dedicated memory flow controller (MFC), and a 256 KB local store (LS). The LS is high-speed SRAM used to store the SPE’s program code and data. Since an SPE cannot directly access main memory, all read and write operations issued by an SPE are limited to its LS. Data transfer operations between an SPE’s LS and main memory are handled by the MFC. The MFC allows for data transfer operations to be carried out while the SPE performs other tasks. This provides the opportunity for data transfer latencies to be hidden using software controlled multi-buffering.

Although the Cell BE processor includes eight SPEs, only six are available for use by applications running on the PS3. One SPE is disabled to make manufacturing the Cell BE more cost effective. Another SPE is dedicated to performing support tasks for the virtualization layer (or hypervisor) known as Game OS. Each SPE delivers a peak of 25.6 Gflop/s single precision performance at 3.2 GHz, so the six SPEs are capable of a max of $6 \times 25.6 \text{ Gflop/s} = 153.6 \text{ Gflop/s}$.

The PPE consists of the simultaneous multi-threading power processing unit (PPU), a 32 KB instruction/data L1 cache, and a 512 KB L2 cache. The PPE uses the PowerPC 970 instruction set, so it is capable of running existing applications compiled for the PowerPC architecture. The PPE can also directly address hardware resources and has full access to main memory. Thus, the PPE is ideal for running an operating system. Although the PPE is a capable processor for computational tasks, its main responsibility is the distribution and management of tasks for the SPEs.

The Cell BE processor is connected to main memory via the memory interface controller (MIC). In the PS3, the MIC provides a 25.6 GB/s connection to 256 MB of

dual-channel Rambus Extreme Data Rate (XDR) memory. Approximately 200 MB of this memory is available to applications running on the PS3.

The PPE, SPEs, MIC, and input/output device system are connected via the element interconnect bus (EIB). The EIB is a four-ring data communication structure, which supports a total bandwidth of 204.8 GB/s. Each of the components has a 25.6 GB/s connection to the EIB.

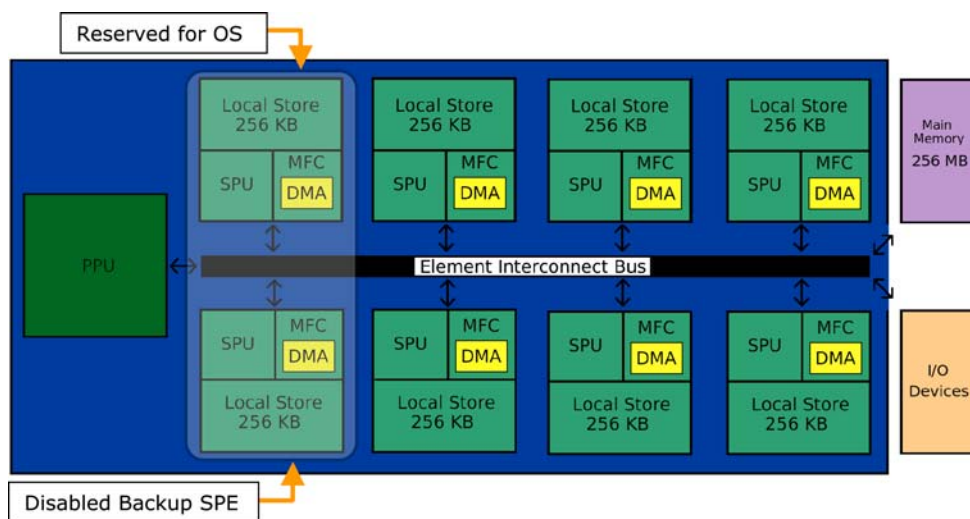
Besides the Cell BE processor and the previously mentioned 256 MB main memory, the PS3 includes a 256 MB nVidia RSX graphics processing unit (GPU), a 60 GB hard drive, a GigaBit Ethernet (GigE) NIC, and a Blue-ray disc drive. The GPU could be used for computation in concert with the Cell BE processor, but at present, its use is blocked by the hypervisor.

Ray tracing

Ray tracing is a visualization technique that uses an eye, a port, an object, and a light source. Part of the reason that we chose to build a ray tracing application, is that it fits well with the requirements of the PS3. Ray tracing does not require a high level of precision, while requiring many computations in a short amount of time. Ray tracing determines if the light source’s rays hit the object for each pixel through the port. This implies that each calculation of hit or miss can be independent. Each SPE can do a line of calculations, until the reflections and lightning have been calculated for the entire screen. The resulting image is then returned to the screen (Fig. 2).

Building a well-developed ray tracer is a very involved project, and we intentionally choose not to do this. This project focused on how the hardware could optimize the ray tracing process, so we choose to work on optimizing the hardware implementation instead of the algorithm. By

Fig. 1 A diagram of the Cell architecture in the PlayStation 3



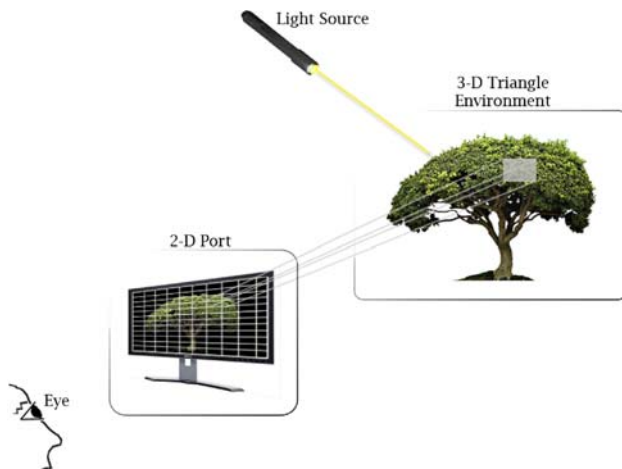


Fig. 2 An illustration of ray tracing

only optimizing the hardware, we can more closely monitor how much gain we are getting from the Cell hardware, and not worry if speed gains are from algorithm or hardware.

We started the project using C code for a simple ray tracing application, developed by Eric Rollins. In his application, he only looked at intersections of circles and rays of light. To expand on this, we first started looking at intersections of triangles and rays. Once we were able to see triangles we went on to modify height field data from tsunami simulations so that they could be visualized. The height field data were large enough where we had to expand on the data handling, since only a limited amount of data can be stored in a LS.

Tsunami visualization on the PS3

PPU program

The following offers a description of the PPU side program of the ray tracing application. First, float values are read from a file containing triangulated data to populate an array of triangle structures in main memory. Each triangle structure represents a single triangle and contains three float vectors. Each float vector represents a point of the triangle and contains three float values for the x , y , and z coordinates of the point. Since each float vector is 16 bytes, each triangle structure is 48 bytes. After the array of triangle structures is filled, a control block structure is initialized for each SPE containing two 32-bit addresses and two integer values. One address gives the location in main memory of a two-dimensional array of long values, referred to from now on as the pixel buffer array. This array is used to store pixel color values given as a result of the ray tracing algorithm. The other address gives the location in main memory to the previously described array of triangle structures. One of the integers gives a value,

referred to here as the line offset value, which dictates how the workload is split between the SPEs. The second integer is a filler value used to simply expand the size of the structure to 16 bytes so that it is the correct size for a DMA transfer. After the control block structures are setup, a thread is created for each SPE using the `spe_create_thread` function from the IBM SDK libspe library. The address of the SPE's associated control block structure is passed as an argument to the `spe_create_thread` function. On creation of an SPE thread, the respective SPU side program is started. A description of the workings of the general SPU side program is given later. Finally, after receiving a mailbox message from the SPEs notifying that all computations were performed, the pixel buffer array is rendered to an image using Simple DirectMedia Layer (SDL).

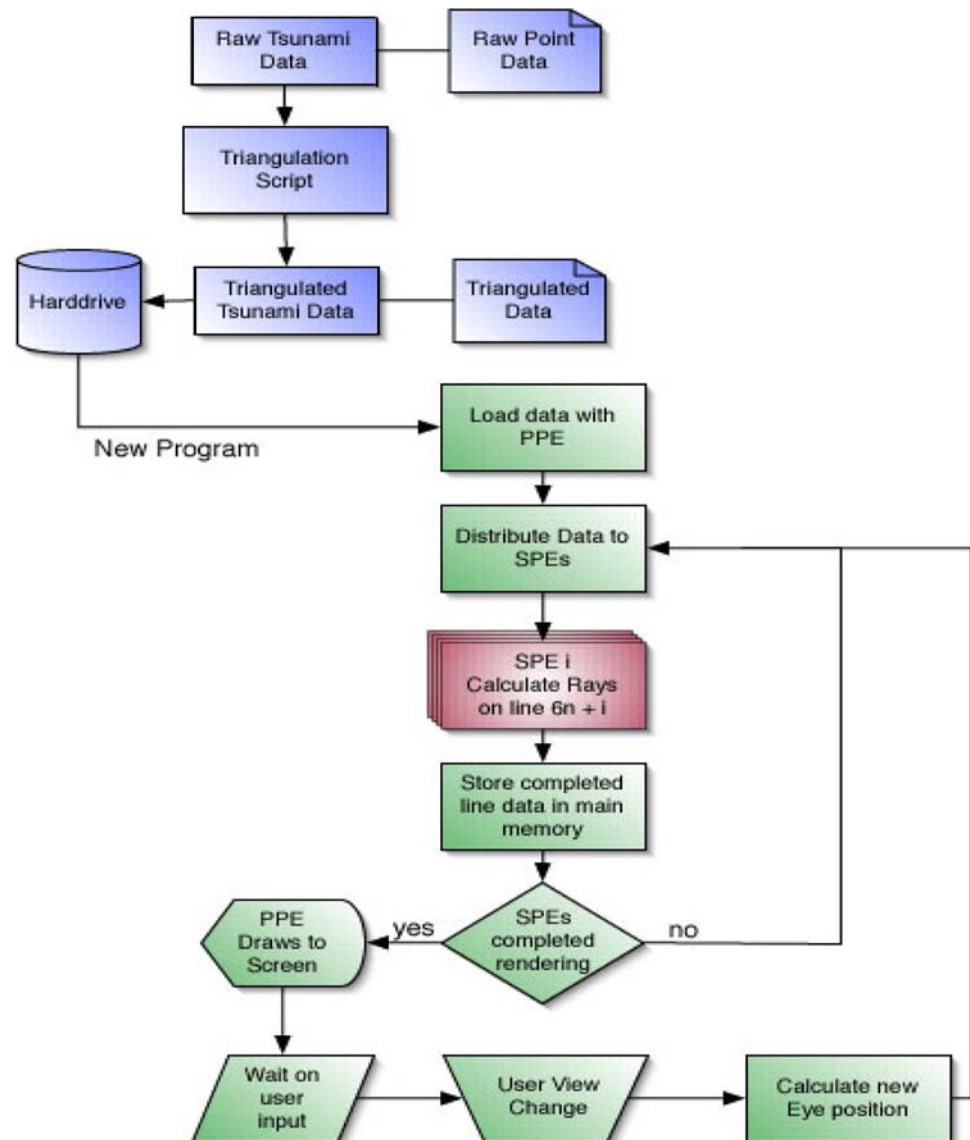
SPU program

The following describes the working of the SPU side program of the ray tracing application. First, the control block structure is transferred from main memory to the LS. After this, the LS is filled with 3,696 triangle structures using an efficient transfer mechanism. This is accomplished through 11 DMA operations where each time 336 triangle structures are copied from main memory to the LS. A DMA operation in which 336 triangle structures are transferred is optimal. This is the case because 336 triangle structures amount to $336 \times 48 \text{ bytes} = 16,128 \text{ bytes}$; no greater whole number of triangle structures can be transferred such that the size of the transfer is under 16 KB (the maximum amount of data that can be transferred) and a multiple of 128 bytes (necessary for an optimal transfer time). A total of 11 of these transfers fills the LS. After the LS is filled with triangle structures, the ray tracing algorithm is executed on those triangle structures, and the pixel buffer array is updated as necessary. The previous two steps, the transfer of triangle structures and the execution of the ray tracing algorithm on the triangle structures, are repeated as needed until all of the triangle structures have been processed. The line offset value specified in the control block structure dictates which lines in the final image of the SPE is responsible for calculating. This ensures that each SPE is responsible for doing 1/6 of the total ray tracing calculations. Last, the updated pixel buffer array is transferred to main memory, and a mailbox message is written to the PPE signaling that computations are complete (Fig. 3).

Parallel layout

The process of developing applications for the Cell BE Processor is more involved than programming for other

Fig. 3 A block diagram of the ray tracing program run on the Cell architecture



common architectures. The design of the Cell BE Processor necessitates an application design consisting of separate programs for the general purpose PPE and the SIMD SPEs. The program running on the PPE is comprised of control code for the application. For example, the PPE program initiates the SPE threads and keeps the SPE programs synchronized. Synchronization and other communication between the PPE and SPE programs are maintained through a mailbox messaging system. The SPEs are vector processors, so special care must be taken to vectorize the programs that run on the SPEs. Libraries developed specifically for the Cell Processor handle vector operations. The SPE programs must explicitly handle the transfer of data between the SPEs and main memory. Each SPE has a 256 KB LS used to hold the program and its data. The PPE program has direct access to main memory, but the SPE programs must use DMA operations to transfer data back and forth from main

memory. The limited memory capacity of the SPE's LS requires the implementation of mechanisms to repeatedly fetch data from main memory during computation. The DMA operations are handled independently from the SPE core by the MFC. The MFCs allow for data transfer times to be hidden in situations where computations associated with a chunk of data are more time consuming than the time taken to transfer the data. Hiding data transfer latencies is accomplished through explicitly implemented buffering techniques. Such techniques range from simple double buffering to software-hyperthreading. For the previously mentioned reasons, the implementation of applications for the Cell Processor clearly requires attention to details beyond those needed in the development of software systems for other common architectures.

The software development kit provided by IBM facilitated programming for the PlayStation and proved to be



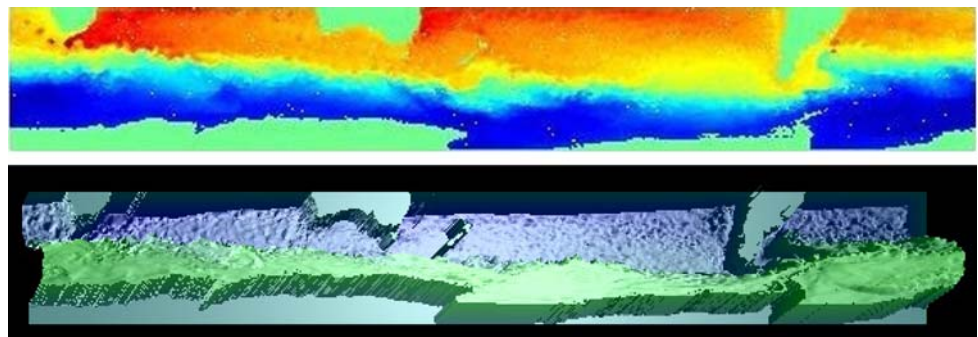
Fig. 4 A render of the triangulated tsunami data

quite helpful is available at <http://www.ibm.com/developerworks/power/cell/>. The IBM SDK included a compiler designed specifically for programs written for the cell, as well as an array of functions which simplified programming. One of the data types available in cell programming is the vector float, which is an array of floats of size four. The SDK allowed us to perform calculations on the entire vector float, instead of forcing us to loop through the array and compute it individually. Other functionality that the SDK provides includes SPE timings and data transfers. However, the SDK did not simplify PPE/SPE communications, and any data passed between the two needed to be extracted using direct memory access (Figs. 4, 5).

Heat equation on the PS3

As an experiment in parallel programming, we tried to choose a simple 2D problem that we would be able to try and run in parallel using the SPUs on the PS3's cell processor. The heat equation ($c\nabla^2 u = \frac{\partial u}{\partial t}$), where u is the temperature field and c is the thermal diffusivity, turns out

Fig. 5 Above a 2D depiction of height field data rendered in MatLab. Below the same height field data rendered as a 3D image in the PS3 ray caster



to be pretty simple when you look at the steady state case where $\frac{\partial u}{\partial t} = 0$. When we solve this equation to figure out how to find $u(i, j)$ we find that we are averaging the surrounding nodes to find the value of the center node. In the Fig. 6, we solve the steady state heat equation by using the recursive equation $x = \frac{a+b+c+d}{4}$ at each iteration.

The tricky part for parallel computation is in the domain decomposition. We get around this by splitting the region into layers for each SPU then share the boundaries between each layer. The boundary needs to contain two nodes to allow the layers to communicate with each other. One of the difficulties we ran into is answering the question of how to solve the problem of communicating between the nodes. Poor communication results in impaired performance, so we spent a lot of time in looking at efficient ways to communicate between SPU nodes without using the PPU. To keep things efficient, we found it was best for the SPUs to directly access values from the memory and the PPU would simply be used to coordinate when data was ready to transfer, instead of actually doing the data transfer (Fig. 7).

Set up of the driven cavity problem

Overview

Problems in computational fluid dynamics (CFD) may prove to be a better application for the PS3 Cell BE processor than ray tracing. The nature of such problems, generally, is grid generation coupled with ODE and PDE solutions through iteration and convergence methods. Thus, the problem can be split up and each processor assigned to a section of the grid. Because CFD focuses more on computation rather than generating high quality visualization, these problems may be more suited for our experimentation on the PS3. The driven cavity problem was chosen to test CFD on the PS3. With the particular parameters chosen, computation on a standard desktop CPU can give a convergence time on the order of 30 min, which leaves room for improvement on a parallel processor.

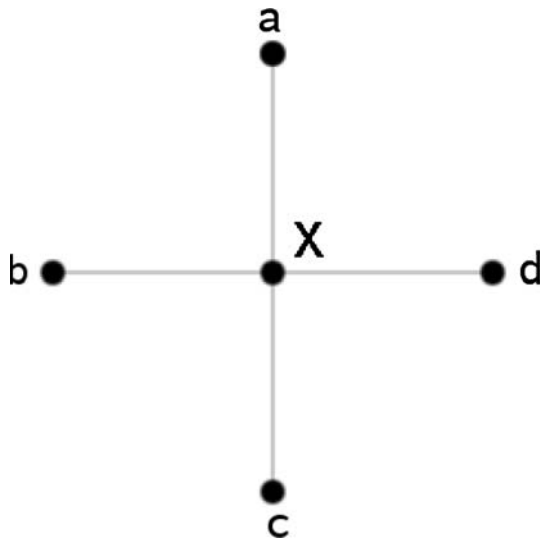


Fig. 6 Central differencing cross: Essentially we are taking the average of these points to compute the next iteration

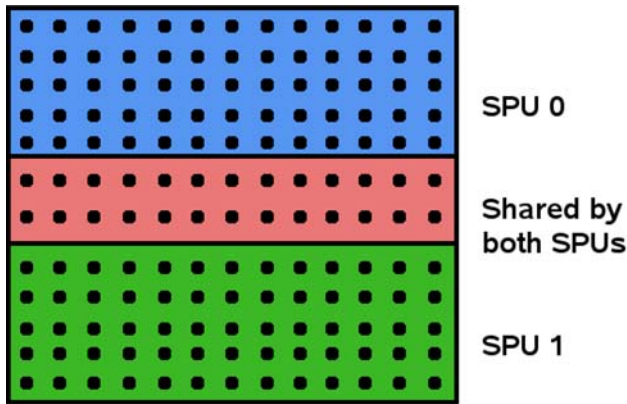


Fig. 7 Domain decomposition between SPUs

The driven cavity problem is well known within the field of CFD and provides a good premise for testing methods as well as studying the effects of input parameters. The problem consists of a rectangular cavity with an upper plate, which moves to the right at a velocity of u_0 . This moving plate drags the fluid to the right and sets up a convection pattern in the cavity. The purpose of this section is to explain this problem and how we would implement it on the PS3.

Theory

The governing theory behind the driven cavity problem is the vorticity–stream function formulation in two-dimensional flow. For two-dimensional flow, a function called the stream function, which satisfies the continuity equation, is defined by:

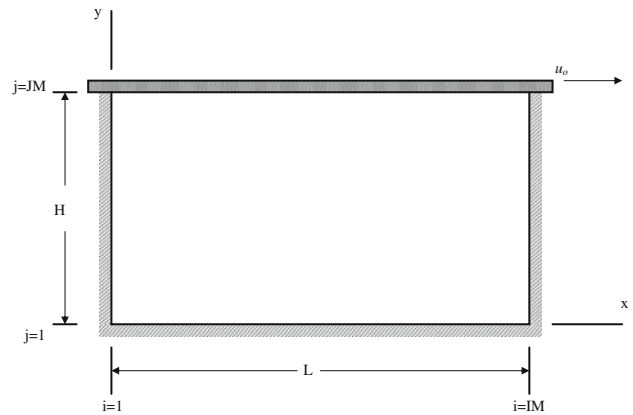


Fig. 8 The domain of solution for the driven cavity problem

$$u = \frac{\partial \psi}{\partial y} \tag{1}$$

$$v = -\frac{\partial \psi}{\partial x} \tag{2}$$

The lines of constant Ψ represent streamlines; the difference between the streamlines gives volumetric flow. To find vorticity, the vorticity transport equation can be derived from the primitive formulation of the incompressible Navier–Stokes equations and is shown in Eq. 3.

$$\frac{\partial \Omega}{\partial t} + u \frac{\partial \Omega}{\partial x} + v \frac{\partial \Omega}{\partial y} = \nu \left(\frac{\partial^2 \Omega}{\partial x^2} + \frac{\partial^2 \Omega}{\partial y^2} \right) \tag{3}$$

Finally, streamline values are calculated from the stream function equation, which is derived from the definition of vorticity along with Eqs. 1 and 2. That is:

$$\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} = -\Omega \tag{4}$$

From these four equations, the behavior of the fluid in the cavity can be calculated by applying the correct numerical algorithm.

Method

The cavity in this problem has dimensions of L by H and the fluid inside the cavity has a kinematic viscosity of $0.0025 \text{ m}^2/\text{s}$. The setup for the cavity is shown in Fig. 8.

The numerical algorithm for solving the governing equations is to use the FTCS explicit scheme and the point Gauss–Seidel formulation to solve for the vorticity and the stream function equations, respectively. The iterative nature of the Gauss–Seidel formulation and the steady state requirements on the vorticity equation requires the use of convergence values for both formulations. A value of 0.001 is set for the convergence of the stream function and a

value of 0.002 is set for the convergence of the vorticity equation. Table 1 provides a summary of the specific input parameters.

The FTCS explicit scheme applied finite difference formulation to Eq. 3 to obtain,

$$\Omega_{ij}^{n+1} = \Omega_{ij}^n + \Delta t v \left[\left[\frac{\Omega_{i+1,j}^n - 2\Omega_{ij}^n + \Omega_{i-1,j}^n}{(\Delta x)^2} + \frac{\Omega_{i,j+1}^n - 2\Omega_{ij}^n + \Omega_{i,j-1}^n}{(\Delta y)^2} \right] - u_{ij}^n \frac{\Omega_{i+1,j}^n - \Omega_{i-1,j}^n}{2\Delta x} - v_{ij}^n \frac{\Omega_{i,j+1}^n - \Omega_{i,j-1}^n}{2\Delta y} \right] \quad (5)$$

for which the vorticity at the next time step can be found. The numerical scheme used to solve the stream function was the Gauss–Seidel formulation, which yields the equation,

$$\psi_{ij}^{k+1} = \frac{1}{2(1 + \beta^2)} \times \left[(\Delta x)^2 \Omega_{ij}^{n+1} + \psi_{i+1,j}^k + \psi_{i-1,j}^k + \beta^2 (\psi_{i,j+1}^k + \psi_{i,j-1}^k) \right]$$

where

$$\beta = \frac{\Delta x}{\Delta y}. \quad (6)$$

These solutions are implemented by solving first for the vorticity within the domain, which is then updated in Eq. 6 and solved for the stream function. This procedure is repeated until convergence is met. One detail to note is that the boundary conditions for the vorticity must be updated as Ω is updated.

Future prospects

The unique architecture of the PS3 offers a few optimization methods that could be pursued in the future. The first method considered utilizes the built-in libraries available in

the cell processor that are capable of SIMD mathematical operations. Once implemented, SIMD instructions can operate on four sets of data at once. This is done by vector operations where each entry in a vector of length four can be added, subtracted, multiplied or divided by the corresponding entries in another vector of length four.

The second method for code optimization is buffering or pipelining. This technique would prompt the SPEs to load an initial set of data, begin computation on that piece of data while a new set is loaded in. Such an algorithm would eliminate much of the time wasted by an inactive SPU as illustrated in Fig. 9.

A third method for optimization is implementing message passing interface (MPI) across more than one PS3. MPI would enable one PPE to send instructions to more than one cell processor. This would increase the number of SPEs available for use, thus decreasing computation time.

Along with MPI, research into rendering on a platform with a graphics card after completing computation on the Cell is being considered. This will resolve any visualization issues.

Another prospect for future work is to switch concentration from the ray tracing algorithm to CFD applications. This is practical because graphics cards can handle most of the issues arising from the ray tracing algorithm and an algorithm that is capable of running a CFD problem would be of higher value.

Final remarks

Programming on the PS3 has posed a very interesting and challenging problem to our group and a lot of ideas and speculation has emerged from our study. Overall, the PS3 is a powerful machine, but much care must be taken when programming for the Cell BE processor. The architecture of the Cell only allows the SPU to work on small amounts of data at a time that leads to great amount of data being moved through memory, and this eventually became a limiting factor in the algorithms used. This limiting factor

Table 1 An overview of parameters to be studied, note that some solutions may not be stable enough to converge

Dimensions and properties	Grid size	
$L = 40$ cm	$\Delta x, \Delta y = 0.01$ m	IM = 41
$H = 30$ cm, 10 cm		JM = 31
$u_o = 5$ m/s	$\Delta x, \Delta y = 0.005$ m	IM = 81
$\Delta t = 0.001$ s		JM = 61
$\nu = 0.0025$ m ² /s	$\Delta x, \Delta y = 0.0025$ m	IM = 161
0.00025 m ² /s		JM = 121
0.000025 m ² /s	$\Delta x, \Delta y = 0.00125$ m	IM = 321
		JM = 241

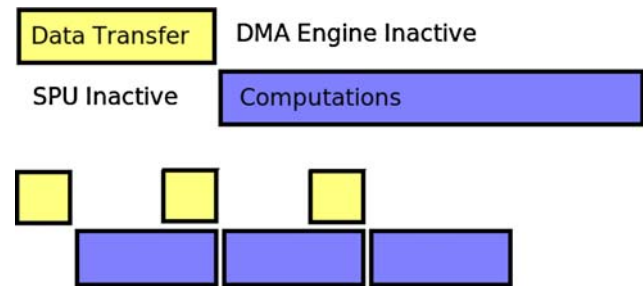


Fig. 9 A depiction of double buffering

requires many levels of optimization in order to bring computation time down to a minimum. Such unique characteristics bring about a tradeoff between time invested in programming and money invested in an architecture that is easier to optimize.

Acknowledgments We acknowledge discussions with Prof. Paul Woodward and help from Spring Liu. This research was supported by the Middleware grant of IF program of National Science Foundation.

References:

- Bellens P, Perez JM, Badia RM, Labarta J (2006) CellSs: a programming model for the Cell BE architecture. Proceedings of the ACM/IEEE SC 2006 conference
- Benthin C, Wald I, Scherbaum M, Friedrich H, (2006) Ray tracing on the Cell Processor. IEEE symposium on interactive ray tracing 15–23
- Buttari A, Dongarra J, Kurzak J (2007) Limitations of the PlayStation 3 for high performance cluster computing, Tech rep, Department of Computer Science, University of Tennessee, UT-CS-07-597
- Buttari A, Luszczyk P, Kurzak J, Dongarra J, Bosilca G (2007) SCOP3: a rough guide to scientific computing on the PlayStation 3, Tech rep, Innovative Computing Laboratory, University of Tennessee Knoxville, UT-CS-07-595
- Dally WJ, Hanrahan P, Erez M, Knight TJ, Labonté F, Ahn J-H, Jayasena N, Kapasi UJ, Das A, Gummaraju J, Buck I (2003) Merrimac: supercomputing with streams, SC2003. Phoenix, Arizona
- IBM (2006) Cell broadband engine architecture, version 1.01
- IBM (2007) Synergistic processor unit instruction set architecture, version 1.2, 2007
- Kurzak J, Buttari A, Luszczyk P, Dongarra J (2008) The PlayStation 3 for high-performance scientific computing, *Computing in Science and Engineering* 84–87
- Williams S, Shalf J, Olike L, Shoal K, Husbands P, Yelick K (2006) The potential of the cell processor for scientific computing. CF '06: Proceedings of the 3rd conference on computing frontiers 9–20