Scientific Programming on the Cell using ALF

J. Hill
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

Abstract
The Cell processor is a heterogeneous multi-core processor, which was originally designed for media applications but potentially applicable to high performance scientific computing applications also, due to its exceptionally high peak floating point capability of over 200GFlops per chip. Unfortunately, programming the Cell is not as simple as copying over your code and compiling, if the full potential of the cell is to be realised. This report looks at a single programming paradigm designed for the Cell, known as Accelerated Library Framework (ALF). A simple scientific application, which shares characteristics of real scientific code currently being run on HPCx, is ported to the Cell using ALF and the performance evaluated. The tools used and the optimisations performed are detailed. The final performance results are disappointing, but this is due to data transfer mechanisms within the ALF programming paradigm, not the Cell. Other studies have shown exceptional performance for scientific tasks on the Cell processor, so while this study shows that ALF is not necessarily the best choice of methods, the Cell is certainly a viable platform for scientific codes in the future.

This is a Technical Report from the HPCx Consortium.

Report available from:
http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0708.pdf

© UoE HPCx Ltd 2003

Neither UoE HPCx Ltd nor its members separately accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>The Cell Processor</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Cell Compilers</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Cell Simulator</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Introduction to ALF</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Hello</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Image Processing</td>
<td>10</td>
</tr>
<tr>
<td>4.1</td>
<td>Naïve Version</td>
<td>11</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Performance</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Improving performance</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Accelerator-side partitioning</td>
<td>15</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Performance</td>
<td>16</td>
</tr>
<tr>
<td>5.2</td>
<td>SIMD optimisations</td>
<td>17</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Performance</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>References</td>
<td>22</td>
</tr>
</tbody>
</table>
1 Introduction

The Cell processor is a heterogeneous multi-core processor [1], originally designed for media applications but potentially applicable to high performance scientific computing applications, due to its exceptionally high peak floating point capability of over 200GFlop/s per chip [1-4]. Cell processors may be used to build high performance parallel systems, in a blade server form factor, such as the RoadRunner petaflop machine currently under construction [5]. Forthcoming Cell systems will provide 16.8TFlop/s peak in double precision from a standard 42U rack, a higher peak performance than a BlueGene/P cabinet [6, 7].

However, the exploitation of this potential is obstructed somewhat by difficulties in programming the Cell. These issues are primarily the result of the different memory model provided on the Cell compared with conventional multi-core processors. Rather than providing all processor cores with access to the same main-memory address space, the Cell has a hierarchical memory architecture where some processing cores have a small private memory called a local store (LS) [8]. Movement of data to and from local stores needs to be explicitly controlled, which provides independence from potentially sub-optimal hardware caching algorithms [9], but is more demanding on programmer.

This report is limited in scope to single-node programs, but aims to demonstrate how to go about programming scientific-like software for the Cell using one programming paradigm, the Accelerated Library Framework (ALF), which was designed to make programming hierarchical memory architectures easier. This document first summarises the Cell architecture and programming techniques, then looks at a simple ALF program - hello. It then takes this forward to develop a simple scientific application and tests the performance and details how the optimisations were applied to the code.

2 The Cell Processor

The Cell processor is a multi-core processor consisting of one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs)1. Typically, an application runs its main thread of execution on the PPE, while executing certain computationally intensive procedures on the SPEs. The PPE is a conventional PowerPC processor, including a 512kB L2 cache and a VMX unit which provides SIMD operations. The SPEs (synonymous with SPU) are SIMD processors, optimised for floating point operations on 128b vectors. Each SPE has a 256kB local store (LS) and associated Direct Memory Access (DMA) engine capable of transfers between LS and main memory, as well as transfers between to and from the LS of another SPE. The SPE cores are relatively simple, using in-order execution with no branch prediction. As a result of this simplified design, an SPE occupies relatively little silicon (Figure 1), each SPE (including LS) uses approximately half the area of the PPE core alone. Not only does this make it possible to fit many SPEs onto a chip, it means that the power consumption of the SPEs is low, resulting in a low Watts/Flop ratio: a key feature of the Cell processor design.

---

1 There is a lot of terminology associated with the Cell processor; even the name itself. The Cell is often referred to Cell Broadband Engine (Cell/B.E. or CBE), Cell processor, or simply “the Cell”. In this report “Cell processor” is preferred, and “Cell” is used as an adjective or adverb, for example “Cell code” or “Cell programming”. Similarly, the processing units in the Cell are referred to as either SPU (Synergistic Processing Unit) and PPU (Power Processing Unit) or SPE and PPE. The SPE is defined to encompass a core, LS and DMA engine. SPU is also used, to refer to only the core and LS. The distinction between SPE and SPU is rarely relevant. The same applies to the PPU.
Processing elements (PEs) are connected via the Element Interconnect Bus (EIB), which provides a peak bandwidth of 204.8GB/s (Figure 1 and Figure 2). Each EIB port has a bandwidth of 25.6GB/s, including the port of the Memory Interface Controller (MIC). This means that while up to 204.8GB/s may be transferred between PEs, the peak main memory bandwidth available to each PE is effectively 25.6GB/s divided by the number of PEs simultaneously competing for access to main memory. The same principle applies to the I/O Interfaces. The PPE cache accesses main memory via the EIB just as the SPEs do. In certain circumstances a DMA “get” to an SPE’s LS may be fulfilled by the PPE or another SPE to avoid going via main memory [6]. Each PE has its own DMA engine, such that a “Get” always refers to loading a value and a “Put” always refers to sending data from that PE to another location. As indicated in Figure 2, the I/O interface IOIF0 may be connected to the equivalent port of another Cell chip to effectively merge the EIBs of both Cells into one, affording the PEs on both chips mutual access to one another without going via main memory. In this setup, it is possible to run one thread on the PPE of one chip, and have it access 16 SPEs without being aware that half of those are on another physical chip.
2.1 Cell Compilers

There are currently two compilers for the Cell processor, xlc and gcc [8]. Each is actually made up of two separate compilers, once each the SPE and PPE. The object files created by the two compilation processes can be incorporated into a single executable file.

2.2 Cell Simulator

Shortly after the inception of the Cell processor, IBM created a Full-System Simulator model for the processor and system architecture that would support software simulation and performance testing [11]. Internally referred to as "Mambo", the simulator allows great understanding of the Cell hardware, down to cycle level instruction performance. The simulator is a free download from IBM and allows anyone to sample Cell programming, without purchasing Cell hardware. The simulator is also an invaluable tool for optimising and understanding code on the Cell, which is described in more detail in section 5.

3 Introduction to ALF

The ALF (Accelerated Library Framework) API is a set of functions to solve parallel problems on multi-core memory hierarchy systems, such as the Cell [12]. ALF uses a host processor and a number of accelerator processors. For the Cell, the host processor is the PPE, with the SPEs acting as accelerators.

ALF provides a set of functions such that an application developer can call accelerated libraries without too much knowledge of the inner workings - i.e. they could use alf_fft(&data), say. The accelerated library developer can develop the library without much knowledge of the computer kernel - s/he just needs knowledge on what data to pass to the kernel and how to partition this data up. The kernel developer can focus on hand-optimised Cell code within the kernel.

Essentially ALF breaks a problem up into separate “tasks”. A workspace is set up and the tasks added to them. The tasks can be run in any order and no order is guaranteed. ALF handles the transfer of data to-and-from the accelerators via a series of buffers. There are two separate buffers; a task context buffer that is available to all accelerators and work block buffer that contains data unique to each accelerator. The work block buffer is automatically “double buffered” if it is less than 120kb in size (this is “hard-wired” into the Cell version of the ALF library and assumes 256kb local store, with 16kb for the computer kernel and two buffers, each of 120kb). The input and output buffers can also be overlapped in order to save space.

The division on the data for each work block can be carried out from the host processor or the accelerators. The data is placed into the buffer as pointers to memory and the details of the transfer are handled to an extent by the ALF runtime library. ALF also has multi-use work blocks where the same portion of data is iterated over on the accelerator.

ALF was chosen for this project as it is a much simpler way of programming the Cell processor (than for example, libspe [13]), but yet is capable of performing as well for certain algorithms [10]. However, ALF is far from the ideal model for the simple task used in section 4 due to the data transfer that must occur, but it nevertheless serves as a useful method of programming the Cell and exploring how to optimise and assess code performance on the Cell.
3.1 Hello

An ALF program is split into two: the host code and the accelerator code (ppu and spu respectively). This is best set up using the following file layout shown in Figure 3.

![Folder hierarchy of the Hello Word program. This setup is used in most of IBM's sample programs.](image)

For the hello program we have the following Makefiles (Box 1 and Box 2), modified from the IBM simulator examples. Note that they require the setting up of CELL_TOP, which points to the simulator directory.

```makefile
1. ########################################################################
2. #   Subdirectories
3. ########################################################################
4. 5. DIRS  := spu
6. 7. ########################################################################
8. #                       Target
9. ########################################################################
10. 11. PROGRAM_ppu  = hello
12. 13. ########################################################################
14. #                       Local Defines
15. ########################################################################
16. 17. IMPORTS = -lmisc -lspe2 -lsync -lm -lalf spu/hello_spu.a
18. 19. ########################################################################
20. #   make.footer
21. ########################################################################
22. ifdef CELL_TOP
23. include $(CELL_TOP)/make.footer
24. else
25. include ../../../make.footer
26. endif
```

Box 1: Makefile for the main (host or ppu) program. Note the reference to the spu DIR and the IMPORTing of the spu/hello_spu.a library.
Box 2. Makefile for the accelerator (spu) program. The output of this compilation is an embedded library which is included in the main program by the PPU makefile.

The main code (Box 3 to Box 8) consists of the usual C headers, along with headers for ALF (line 8). An important bit of code here is on line 13. This is the SPE code. Notice that the variable here is the same as the target in the SPE Makefile above (Box 2).

Box 3: The header section of the main code.

Next comes the standard C main function. Inside we can define the ALF task variables:

Box 4: ALF task variables.

Here we set up some standard variables \((i, rc \text{ (errors)} \text{ and } \text{node})\) along with the ALF variables. \(\text{parm}\) is a struct that is used to pass parameters to the kernel. \(\text{half}\) is a handle to one instance of the ALF runtime. It is created by \(\text{alf_init}\) (see later) and destroyed by \(\text{alf_exit}\) (see even later). \(\text{htask}\) is a task handle onto which work blocks are attached. This is created later with \(\text{alf_task_create}\) and can be destroyed with \(\text{alf_task_destroy}\) or is destroyed implicitly by \(\text{alf_exit}\) is called. \(\text{hwb}\) is a handle to a work block, which provides the data to an accelerator to work on. \(\text{alf_task_info_t}\) is a data structure that holds information on the task list to be given to the accelerators, such as the
size of the I/O buffers. alf_task_info_t_CBEA is the same structure specialised for the Cell.

```c
24. /* begin alf routine */
25. alf_configure(NULL);
26. rc = alf_query_system_info(ALF_INFO_NUM_ACCL_NODES, &nodes);
27. if (rc < 0) {
28.   fprintf(stderr, "Failed to call alf_query_system_info.\n"");
29.   return 1;
30. } else if( nodes <= 0 ) {
31.   fprintf(stderr, "Cannot allocate spe to use.\n");
32.   return 1;
33. }
34. rc = alf_init(&half, nodes, ALF_INIT_PERSIST);
35. if( rc <= 0) {
36.   fprintf(stderr, "Cannot init ALF libary(%d).\n", rc);
37.   return rc;
38. }
39. spe_tsk.spe_task_image = &hello_spu;
40. spe_tsk.max_stack_size = 4096;
41. /* filling out the task info structure */
42. memset(&tinfo, 0, sizeof(tinfo));
43. tinfo.p_task_info = &spe_tsk;
44. tinfo.task_context_buffer_read_only_size=0;
45. tinfo.task_context_buffer_writable_size=0;
46. tinfo.parm_ctx_buffer_size = sizeof(parms_t);
47. tinfo.overlapped_buffer_size = 0;
48. /* this task is set to generate DMA lists on the control node */
49. tinfo.task_attr = 0;
```

Box 5: Creating the ALF tasks

Once the variables are allocated, the ALF tasks can be constructed. This involves the alf_configure routine which must be called in every ALF program. The single parameter is an implementation specific data pointer which is not required for the Cell hence is NULL. The program then gets the number of accelerators available using alf_query_system_info. The first parameter allows the querying of different parameters, in this case the number of accelerators. On a dual Cell-Blade system, it correctly reports 16 SPUs. Other parameters that can be queried include host and accelerator memory and host and accelerator memory alignment. The ALF is then initialised with alf_init, which requires the alf handle, the number of accelerators to use, and a parameter to say what to do in the case there are not enough available accelerators. In this case we wait until all are available. Other cases can be to ALF_INIT_COMPROMISE (take as many as possible up to the number given) or ALF_INIT_TRY (return an error if the number cannot be satisfied).

The next stage is to fill in the task structure with information about buffers and the SPU program to use.

```c
43. /* filling out the task info structure */
44. memset(&tinfo, 0, sizeof(tinfo));
45. tinfo.p_task_info = &spe_tsk;
46. tinfo.task_context_buffer_read_only_size=0;
47. tinfo.task_context_buffer_writable_size=0;
48. tinfo.parm_ctx_buffer_size = sizeof(parms_t);
49. tinfo.overlapped_buffer_size = 0;
50. /* this task is set to generate DMA lists on the control node */
51. tinfo.task_attr = 0;
```

Box 6: Setting up the ALF environment.

This section tells the accelerators which code to run (hello_spu) and sets up the buffers, in this case 0 as we have no buffers. We also set up the size of parameters we are passing to the work blocks (line 52).
/* creating task */
rc = alf_task_create(&htask, half, &tinfo);
if (rc < 0) {
    fprintf(stderr, "Cannot create ALF task(%d).\n", rc);
    alf_exit(half, ALF_SHUTDOWN_FORCE);
    return 1;
}

/* wb create and add param and io buffer */
for(i=0; i<NTASKS; i++) {
    alf_wb_create (hwb, htask, ALF_WB_SINGLE, 0);
    /* pass i and 10 to spu */
    parm.v = i;
    parm.h = 10;
    alf_wb_add_param (hwb, &parm, sizeof(parm),
    ALF_DATA_BYTE, 0);
    alf_wb_enqueue(hwb);
}

/* alf task wait until wb processed */
alf_task_wait(htask, -1);
alf_task_destroy(&htask);

printf ("All done!\n");
return 0;

#include <alf_accel.h>
#include <stdio.h>
#include ../hello.h

int alf_comp_kernel(void *p_task_context,
    void *p_parm_ctx_buffer,
    void *p_input_buffer,
    void *p_output_buffer,
    unsigned int current_count,
    unsigned int total_count )
{
    parms_t *p_parm = (parms_t *)p_parm_ctx_buffer;
    printf("I'm an accelerator and have the numbers %d and %d\n", p_parm->h, p_parm->v);
    return 0;
}
4 Image Processing

To test ALF more thoroughly, a more realistic problem was attempted. In this example, an image (e.g. Figure 4) which has undergone simple edge detection is reconstructed to its original state. This is a simple Jacobi algorithm to solve a 2D Poisson equation, requiring many iterations over the data to achieve the end result. Using MPI programming a one-dimensional decomposition can be used, requiring data to be passed to the neighbouring processors in order to parallelise the problem. For ALF, the data can again be broken into individual blocks, enabling a simple one-dimensional parallelisation.

![Figure 4: Original image and the result of the simple edge detection.](image)

The edge image is constructed using the simple equation:

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4image_{i,j}$$

(1)

The image can then be reconstructed from the edge using numerous repetitions of the following:

$$new_{i,j} = \frac{1}{4}(old_{i-1,j} + old_{i+1,j} + old_{i,j-1} + old_{i,j+1} - edge_{i,j})$$

(2)

Although the code required to do this is very simple, it serves the purpose of comparing a typical, scientific program (i.e. many iterations over an array) to the equivalent programmed in ALF for the Cell processor.

The image will be split into horizontal slices and each slice will be encoded as a work block. ALF will then ship a work block to a free SPU, which will process that segment. After all work blocks are processed, the new variable needs to be stored in the old variable for the next iteration. This is equivalent of “halo-swapping” in the MPI code, except the whole data must be passed back to the PPU for synchronising, ready for the next iteration. ALF provides a callback function to do such synchronisation. In order to have the necessary data for equation (2), it is necessary to pass a block of old which has a halo around it to accommodate the requirement for values in adjacent cells that would otherwise be in the neighbouring block. The code also requires the original edge data but this does not require a halo. The output buffer contains a single block (with no halo) of the variable new.

Three versions of the code will be created; a naïve version which aims to get the code up and running as quickly as possible; an accelerator-partitioned version, which aims to offload as much work as possible to the SPEs; and a SIMD optimised version, which aims to increase performance via SIMD operations wherever possible.
4.1 Naïve Version

The code for this is very similar to that above in the simple “Hello World” example. Work blocks are set up and stored to the task. However, we also pass data to the work blocks in this instance. There is an additional variable \((\text{hsync})\) compared to the simple “Hello World” code to handle the synchronisation:

```c
34. alf_handle_t              half;
35. alf_task_handle_t         htask;
36. alf_wb_handle_t           hwb;
37. alf_task_info_t           tinfo;
38. alf_task_info_t_CBEA      spe_tsk;
39. image_parms_t             parm;
40. alf_wb_sync_handle_t      hsync;
```

**Box 10: Variables required for handling the ALF task.**

There are two buffers: input and output. The input buffer will contain a block \((\text{parm.h} \times \text{parm.v})\) of the old variable and a block of the original edge image. The size of these buffers must be set correctly, using the following code:

```c
41. tinfo.input_buffer_size = H*V*sizeof(float) + parm.h*parm.v*sizeof(float);
42. tinfo.output_buffer_size = parm.h*parm.v*sizeof(float);
```

**Box 11: Setting the buffer sizes.**

In order to allow ALF to transparently handle the movement of data from the PPU to the SPU, memory must be aligned at 128 bytes. This is done using the following macro.

```c
#define MY_ALIGN(_my_var_def_, _my_al_)  _my_var_def_ __attribute__((__aligned__(_my_al_)))
```

Any aligned arrays are then declared as:

```c
MY_ALIGN(float masterbuf[M][N], 128);
```

The bulk of the code in the main program is to create the work blocks. This is very similar to the first example, but in this case we pass data to the SPUs using the ALF buffers. The loop below (Box 12) iterates over the number of computational iterations. For each of these a work block is created and the parameters (in this case the size of the block) are attached. The image is then broken into horizontal strips of size \(H\). Each strip is attached to the input buffer and the equivalent strip of the original edge data (without halos) is also attached. The output buffer is then attached. Finally the work block is queued.

At the end of each iteration, the synchronisation is performed by the call to `alf_wb_sync`. The function arguments contain a call to the function `swap_halos`, which is called after each iteration. This function simply copies the new values to `old`, ready for the next iteration.
for (iter=1;iter<=MAXITER; iter++) {
    /* create wb and add param and io buffer */
    for(i=0; i<M; i+=parm.v) {
        alf_wb_create (&hwb, htask, ALF_WB_SINGLE, 0);
        rc = alf_wb_add_param (hwb, &parm, sizeof(parm), ALF_DATA_BYTE, 0);
        print_error(rc, "error allocating params", &half);
        rc = alf_wb_add_io_buffer(hwb, &masterbuf[i][0],
            parm.h*parm.v, ALF_DATA_FLOAT, ALF_BUFFER_INPUT);
        print_error(rc, "error allocating buffer 1", &half);
        rc = alf_wb_add_io_buffer(hwb, &old[i][0],
            H*V, ALF_DATA_FLOAT, ALF_BUFFER_INPUT);
        print_error(rc, "error allocating buffer 2", &half);
        rc = alf_wb_add_io_buffer (hwb, &new[i][0],
            parm.h*parm.v, ALF_DATA_FLOAT, ALF_BUFFER_OUTPUT);
        print_error(rc, "error allocating output buffer", &half);
        alf_wb_enqueue(hwb);
    }
    alf_wb_sync(&hsync, htask, ALF_SYNC_BARRIER, &swap_halos, NULL, 0);
}

Box 12: Setting up the work blocks.

On the SPU, the code is very simple. The two buffers are passed as pointers to blocks of memory. Using pointer arithmetic, the input buffer is separated into two and equation (2) is implemented as a nested loop.

```c
#include <alf_accel.h>
#include "../image.h"
#include <stdio.h>

int alf_comp_kernel(void *p_task_context,
    void *p_parm_ctx_buffer,
    void *p_input_buffer,
    void *p_output_buffer,
    unsigned int current_count,
    unsigned int total_count ) {
    unsigned int i, j;
    float *sa, *sb, *masterbuf;
    unsigned int h, v;
    image_parms_t *p_parm = (image_parms_t *)p_parm_ctx_buffer;
    h = p_parm->h;
    v = p_parm->v;
    masterbuf = (float *) p_input_buffer;
    sb = (float *) p_output_buffer;
    sa = masterbuf + (h)*v;
    sa = sa + (h+2)+1;
    for (i=0;i<v;i++) {
        for (j=0;j<h;j++) {
            sb[i*h+j] = 0.25 * (sa[(i+1)*(h+2)+j] + sa[(i-1)*(h+2)+j] +
                sa[i*(h+2)+j+1] + sa[i*(h+2)+j-1] -
                masterbuf[i*h+j]);
        }
    }
    return 0;
}
```

Box 13: The SPU code
4.1.1 Performance

Performance on the Cell processor can be measured in two ways; using standard techniques (such as clock functions in C) or using the simulator in cycle-accuracy mode.

Placing timing code around the ALF queuing code on the PPU will include times for the setting up of the problem on the SPEs (including data transferral), the compute kernel and the pulling down of the SPE threads. Performance was testing using a variety of data block sizes over 1 to 8 SPEs (i.e. a single Cell chip).

Results for this version of the image code are unsurprising and uninspiring. The code does not scale well nor perform particularly well. There is little difference in wallclock time when altering the data block size, although the larger data block size is slightly quicker (Figure 5). XLC performs slightly better, but the difference is marginal (Figure 6).

![Figure 5. Execution time for the image code compiled using GCC (left) and XLC (right) with various data block sizes for an image of 840x600 pixels. Data block sizes are in bytes and include both input and output data.](image)

In terms of cycles, the code is not particularly inefficient. Using the Cell simulator, it is possible to create a breakdown of the instructions issued and the number of cycles taken to perform those instructions. Performing a single iteration on one SPE in the simulator and then printing the statistics produce the output seen in Box 14, which are the statistics for processing one block of data. A total of 316,454 cycles were consumed for the processing of a block with 188,512 instructions issued (9 of which were “no ops”). Over 55% of cycles were waiting for...
other dependencies to complete. These dependencies are listed in the lower half of the print out. In this case the majority were of type “FP6”: floating point operations.

Given total run times of 10’s of seconds (and a quickest of 9.43s), but only a few hundred thousand cycles (at 3.2Ghz, this is only a fraction of a second of wallclock time) spent doing useful work, the main bottleneck in performance must be the transfer of data. One solution to this is to transfer data using the SPEs, rather than rely on the PPE.

Performance Cycle count 316454
Performance Instruction count 188512 (188503)
Performance CPI 1.68 (1.68)
Branch instructions 6744
Branch taken 6720
Branch not taken 24
Hint instructions 8
Hint hit 6712
Contention at LS between Load/Store and Prefetch 2

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single cycle</td>
<td>94373  (29.8%)</td>
</tr>
<tr>
<td>Dual cycle</td>
<td>47065  (14.9%)</td>
</tr>
<tr>
<td>Nop cycle</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Stall due to branch miss</td>
<td>273 (0.1%)</td>
</tr>
<tr>
<td>Stall due to prefetch miss</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Stall due to dependency</td>
<td>174743 (55.2%)</td>
</tr>
<tr>
<td>Stall due to fp resource conflict</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Stall due to waiting for hint target</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Issue stalls due to pipe hazards</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Channel stall cycle</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>SPU Initialization cycle</td>
<td>0 (0.0%)</td>
</tr>
<tr>
<td>Total cycle</td>
<td>316454 (100.0%)</td>
</tr>
</tbody>
</table>

Stall cycles due to dependency on each pipelines
- FX2 11 (0.0% of all dependency stalls)
- SHUF 20162 (61.5% of all dependency stalls)
- FX3 1 (0.0% of all dependency stalls)
- LS 4 (0.0% of all dependency stalls)
- BR 0 (0.0% of all dependency stalls)
- SFR 0 (0.0% of all dependency stalls)
- LNOP 0 (0.0% of all dependency stalls)
- NOP 0 (0.0% of all dependency stalls)
- FXB 0 (0.0% of all dependency stalls)
- FP6 154560 (88.4% of all dependency stalls)
- FPD 5 (0.0% of all dependency stalls)
- FPD 0 (0.0% of all dependency stalls)

The number of used registers are 21, the used ratio is 16.41

<table>
<thead>
<tr>
<th></th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FX2</td>
<td>11     (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>SHUF</td>
<td>20162  (61.5% of all dependency stalls)</td>
</tr>
<tr>
<td>FX3</td>
<td>1      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>LS</td>
<td>4      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>BR</td>
<td>0      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>SFR</td>
<td>0      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>LNOP</td>
<td>0      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>NOP</td>
<td>0      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>FXB</td>
<td>0      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>FP6</td>
<td>154560 (88.4% of all dependency stalls)</td>
</tr>
<tr>
<td>FPD</td>
<td>5      (0.0% of all dependency stalls)</td>
</tr>
<tr>
<td>FPD</td>
<td>0      (0.0% of all dependency stalls)</td>
</tr>
</tbody>
</table>

Box 14: Output from the “display statistics” command in the Cell simulator. The output gives detailed, cycle level performance figures for the instrumented portion of code.

5 Improving performance

There are several methods available for improving the performance of the above code. Firstly, instead of the PPE being responsible for partitioning the data, the SPEs can partition their own data. Given the small amount of processing that is done on each data block compared to the time taken to transfer data, this may make a noticeable difference. However, another option to improve performance on a Cell processor comes with optimising the compute kernel using SPE SIMD instructions. The main reason for this is to take advantage of the vector nature of the Cell and process four array elements simultaneously. This section looks at accelerator-side partitioning and SIMD-isation of the compute kernel, briefly covering some of the tools available to the Cell programmer.
5.1 Accelerator-side partitioning

Instead of the PPE performing the necessary division of labour, the SPEs themselves can do the task. Performing accelerator-side partitioning involves the creation of two new SPE functions and a few altered lines in the PPE code.

The two SPE functions that must be implemented are:

```
int alf_prepare_input_list(void *p_task_context,  
void *p_parm_ctx_buffer,  
void *p_dt_list_buffer,  
unsigned int current_count,  
unsigned int total_count ) {

and

int alf_prepare_output_list(void *p_task_context,  
void *p_parm_ctx_buffer,  
void *p_dt_list_buffer,  
unsigned int current_count,  
unsigned int total_count ) {
```

which prepare the input and output data respectively.

The data from the host PPE is then pulled across from the PPE to the appropriate SPE (which is the SPE that received his workblock). A typical input function is shown in Box 14. Note the use of effective addresses (ea) which allow the mapping of address space from the PPE to the SPE.

```
1. unsigned int const transfer_size = (1024 * 16) / sizeof(float);
2. unsigned int block_size = p_parm->h*p_parm->v;
3. unsigned int transfers = block_size / transfer_size;
4. unsigned int remainder = block_size % transfer_size;
5. pInput1 = p_parm->edge;
6. pInput2 = p_parm->input;
7. ALF_DT_LIST_CREATE(p_dt_list_buffer,0);
8. ea.ui[0] = 0;
9. for (i = 0; i < transfers; ++i) {
10.   ea.ui[1] = (unsigned int) (pInput1 + (i * transfer_size));
11.   ALF_DT_LIST_ADD_ENTRY (p_dt_list_buffer, transfer_size, ALF_DATA_FLOAT, ea);
12. }
```

Box 15: Typical alf_prepare_input_list code.

On the PPE side, the code needs a few changes, mainly in the set up and queuing of work blocks. First, the PPE needs to be told that the SPEs will be defining their own data for the work blocks. This is done by setting:

```
tinfo.task_attr = ALF_TASK_ATTR_PARTITION_ON_ACCEL;
```

Secondly, the work block code is substantially reduced to:
Box 16: Setting up work blocks using accelerator-side partitioning. Compare to Box 12.

5.1.1 Performance

Despite the decrease in workload for the PPE (instead the same work is split \(N\)-ways over the SPEs, where \(N\) is the number of SPEs used), there is no substantial speed-up for performing accelerator-side partitioning in this example (Figure 7). As with the previous version, xlc is slightly faster and larger workblocks tend to produce faster code. Despite the disappointing performance results, accelerator-partitioned data is useful due to the ability of ALF to perform buffer overlapping and multi-use work blocks, which while not applicable in this case, may be to other cases.
5.2 **SIMD optimisations**

The SPE presents a number of significant differences from a conventional processor, largely stemming from its SIMD design. As the SPE is a specialised SIMD processor, all loads, stores and arithmetic instructions operate on vectors of 128 bits [14]. These vectors store 4 floats, 2 doubles or 8 characters, and so on [15]. This causes a performance hit for conventional scalar code, since to operate on a scalar value (e.g. 1 float), the operation is performed on a whole vector (4 floats) and the result is then picked out, requiring a series of vector rotation operations if the scalar value is not already at the start of the vector. This process results in a high cost per Flop for scalar arithmetic compared to vector arithmetic. The cost of scalar operations is increased further by the fact that there is no scalar load or store instruction, so to store a scalar value it is necessary to load a vector, insert the scalar value and then store the vector. SPE load/store operations transfer vectors between the LS and registers.

The SPE has a large unified register file, containing 128 registers, each of which may store any data type. Load/store operations operate on 16B-aligned LS addresses: to load a vector which is not so aligned it is necessary to load both 16B-aligned addresses which it spans, and then extract the desired elements using a shuffle instruction. As with operations on scalar data, this introduces a substantial overhead.

Vector floating point instructions include the usual add and multiply operations, as well as a fused multiply-add. The fused multiply-add instruction takes the same length of time (6 cycles) to complete a separate add or multiply instruction, so merging add and multiply instructions can double floating point performance for some algorithms (such as matrix-matrix multiplication). However, it may also give a slightly different result, since the result is rounded only once at the end, rather than once after the add operation and once after the multiply operation.

So for example:

```c
oneOverFour = spu_splats(0.25f);
```

places the value 0.25 in each partition of the vector oneOverFour (a total of four values as oneOverFour is a float vector). This can then be multiplied to four floating point values simultaneously, which are held in a vector, using the multiply intrinsic, `spu_mul`:

```c
tmp3 = spu_mul(tmp3,oneOverFour);
```

These intrinsic operations map directly onto SPU instructions. A full list of intrinsic operations can be found in

The SPU has two pipelines, named even (pipeline 0) and odd (pipeline 1), into which it can issue and complete up to two instructions per cycle, one in each of the pipelines. Whether an instruction goes to the even or odd pipeline depends on its instruction type, which is related to the execution unit that performs the function. Loads, stores, branch prediction and shuffles are done on the odd pipeline. Calculations are done on the even pipeline.

When working in single precision with perfectly pipelined code, the SPE is capable of performing one vector multiply-add operation per cycle [1]. This renders a peak performance of 25.6GFlop/s per SPE, for the current 3.2GHz product. It is by multiplying this by 8 that the overall peak of 204.8GFlop/s is derived. Recalling that the bandwidth to main memory is 25.6GB/s, the SPE is easily capable of starving itself of data if it is only performing a few
operations on each item of data loaded. However, once the data is in the LS of an SPE, it can be operated upon very efficiently, since a pipelined load/store instruction may be issued at a frequency of one per cycle, and has the same duration of execution as a floating point multiply-add (6 cycles).

The SPE is a fairly simple processor with no hardware branch prediction. The programmer can place hints into the assembler code using the built-in intrinsic:

```
__builtin_expect(condition, value)
```

For example, a loop condition may benefit from the following hint:

```
__builtin_expect(j<h, 1)
```

Which works well for large values of h as j is nearly always less than h.

In order to make full use of the SIMD intrinsics, it is necessary to re-cast the floating point arrays to vectors. Each vector contains four floating point values, so array dimensions have to be divisible by four. In doing so, the whole of the kernel calculation can be re-implemented as vector operations, which should substantially reduce the number of cycles taken. However, floats have to be aligned correctly (16byte alignment), which ALF takes care of when it does the data transfer, in order to carry out the recast from floats to vector floats. This was straightforward for the \( i-1 \) and \( i+1 \) vectors, i.e.,

\[
\text{sa}[(i+1)*(h+2) + j] + \text{sa}[(i-1)*(h+2) + j]
\]

but not for the \( j+1 \) and \( j-1 \) vectors, i.e.,

\[
\text{sa}[i*(h+2) + j+1] + \text{sa}[i*(h+2) + j-1].
\]

The code makes use of the `spu_shuffle()` function to grab data that isn’t contiguous in memory and place it in vectors ready for the calculation.

Implementing SIMD optimisations only occurred in the computer kernel. The full code is shown below in Box 17. The original operations are included as comments.

There are some very good tools that allow a very detailed look at the code produced by the compiler. The simulator was covered earlier, but there is also a tool, asmVis, that allows visual inspection of the assembler code and rearrangement of that code. This clearly identifies dependencies, which in turn can help eliminate or alleviate them. asmVis is a java application that takes the assembler file (.s) and an option timing file (.s.timing) to produce a visual representation of the code (Figure 8). The tabs at the top of the program give the functions available in the assembler. Vertical 'x's marks clock cycles where instructions were performed. A column of 'x's correspond to the cycles consumed by the instruction that is level with the first 'x'. Dependencies are highlighted using red pipe symbols (Figure 8).
Box 17: The SIMD version of the computer kernel. Note the use of register for variables, making use of the SPE’s abundant registers, and the use of SPU intrinsics, such as spu_mult to complete four floating point operations simultaneously.

```c
const vector unsigned char step_pattern1 =
(12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27);
const vector unsigned char step_pattern2 =
(4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19);
// These assume sa, sb and masterbuf are 16B aligned
vector float *sa_v = (vector float *)sa;
vector float *sb_v = (vector float *)sb;
vector float *masterbuf_v = (vector float *)masterbuf;

oneOverFour = spu_splats(0.25f);

// work out where to start vectors from
vecind_pl = (h4/2)-1;
vecind_ml = -1;
vecind_ = (h4/4)-1;
vecind = 0;

for (i=0; _builtin_expect(i<v, 1); i++) {
    vecind_p1 += 2;
    vecind_m1 += 2;
    vecind_   += 2;
    for (j=0; _builtin_expect(j<h, 1); j=j+4) {
        // this is the "original" function, before SIMDisation
        // sb[i*h+j] = 0.25 * (sa[(i+1)*(h+2)+j] + sa[(i-1)*(h+2)+j] +
        //              sa[i*(h+2)+j+1] + sa[i*(h+2)+j-1] -
        //              masterbuf[i*h+j]);
        tmp1 = sa_v[vecind_pl];
        tmp2 = sa_v[vecind_ml];
        tmp3 = spu_add(tmp1, tmp2);
        tmp4 = sa_v[vecind_ - 1];
        tmp5 = sa_v[vecind_];
        tmp6 = sa_v[vecind_ + 1];
        tmp7 = spu_add(tmp4, tmp5);
        tmp8 = spu_shuffle (tmp7, tmp6, step_pattern1);
        tmp9 = spu_shuffle (tmp8, tmp3, step_pattern2);
        tmp10 = spu_add(tmp9, tmp3);
        tmp11 = spu_add(tmp10, oneOverFour);
        sb_v[vecind] = tmp11;
        vecind++; vecind_pl++; vecind_ml++; vecind_++;
    }
}
```
Figure 8: asmVis view of the computational kernel in Box 17. The two pipelines are clearly seen, with labelled instruction on each side. ‘x’'s mark instructions being performed, with red pipe symbols highlighting dependencies.

The dependencies seen in Figure 8 are due to having to wait for floating point operations. The first is the instruction \( \text{fa} \) (float add, instruction 1684, highlighted in black in Figure 9), which is adding two vectors stored in registers and storing the result in another register. The next instruction, overlaps a little with this (instruction 1685, blue in Figure 9), but a third instruction (1686), another add, depends on the first and second add instructions, hence the execution of this is delayed. By providing this level of detail it is possible to fully understand what the compiler has produced, and given sufficient time, develop a high performing computer kernel.

Figure 9: Close up of instructions seen in Figure 8. See text for details.
5.2.1 Performance

As with previous versions of the code, the performance of the optimised version is disappointing, both in terms of scaling and performance. Clearly this code is not ideally suited to the ALF model as a lot of time is spent getting data to the SPEs where not much processing is then performed on it. In an MPI implementation of the code, the data transfer consists of strips of data to neighbouring processes, whereas here, using the ALF model we have to send out data from the PPU, performs the calculation on that data in the SPU, then fetch data back to the PPU in order to update. The data cannot be left on the SPE at each iteration and data sent to the neighbouring processor when required. Implementing the algorithm in libspe would allow a more flexible approach where the halos could be sent to the neighbouring SPE in a very MPI-like fashion, but would require much more work than time allows. In order to keep the code simple and understandable (except for the kernel) the performance has been sacrificed somewhat.

In terms of cycle level performance, however, adding SIMD intrinsics to the code produced a code that used just 40% of the cycles of the naïve version to process a single block of data (128,232 cycles vs. 316454 cycles). This figure could well be reduced more by unrolling the inner loop by a factor of two, allowing intermingling of the loops and reducing dependencies.

6 Conclusions

The Cell is potentially a very powerful architecture for scientific codes. Although it is sometimes difficult to understand the problem in a Cell-type way, the tools are available to help with this. Tricky issues encountered during this project were data transfer issues, which require a thorough understanding of memory alignment, even when hidden with ALF and debugging cell the code. Although it is possible to attach the debugger to the SPE thread within the simulator, it was not achieved during this project, so debugging was reduced to print statements.
Clearly, given the poor performance results, one might think that the Cell is not ideally suited. However, it is important to note that the ALF method of programming did not suit this problem, something that was not appreciated before the results were obtained. Clearly, shuffling all the data to-and-fro each iteration is inefficient and wasteful. Better results may be obtained using libspe2, which is very low-level, but gives fine-grained control on data movement. In other circumstances, ALF performs nearly as well as libspe2 [10]. Impressive preliminary results are beginning to appear regarding scientific code on the Cell [4, 10, 16]. The number of results like these is sure to increase with time.

This report looks at a single Cell processor, but Cell-based supercomputer are currently being built [5]. The parallel programming paradigm for such machines is very much like the mixed-mode model, where there are multiple MPI processes, each running a small number of threads (e.g. using Open-MP). A large program could be split over multiple Cells PPEs (using MPI for example) and the compute kernels of the code further parallelised over the accelerators where appropriate.

It is also important to remember that the Cell is a very new architecture. The compilers are still in their infancy. This explains many of the intricacies of Cell programming that the programmer must deal with, such as memory alignment, which would hopefully disappear over time as the compiler takes on more of this work. Other technologies, such as RapidMind [17], CellSs [18] and even MPI [19, 20] will make the task of programming the Cell less arduous.

7 Acknowledgements

The author would like to thank Mike McNamee (IBM Edinburgh) and Joachim Jordan (IBM Böblingen) for access to Cell hardware, as well as John Spray for technical help and running the benchmarks.

8 References


