Lattice QCD on the Cell Processor

John C. Spray

August 24, 2007

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2007
Abstract

The Cell processor is a novel heterogeneous multi-core processor, with a large peak floating point performance. The extent to which this high potential performance may be applied to computationally hard scientific problems is not yet well understood. One such computationally demanding area of research is Lattice Quantum Chromodynamics (QCD), an important field of study in particle physics.

In this report, various approaches to porting parts of a Lattice QCD library called QDP++ to the Cell processor are explored. Low-level programming is compared with a workblock abstraction, a software cache, and a high-level data parallel environment. Great variations between these approaches are found to exist both in the performance attainable and the programmer effort required. For large datasets, implementations of a simple BLAS-type operation are found to be limited in performance by memory bandwidth. For smaller datasets, complex variations in performance exist depending on the number of processing elements used and the division of work.

In response to conclusions drawn from performance analysis of the various techniques, a high performance (up to 45 GFlop/s) implementation of the Wilson Dirac operator (Dslash) is implemented, providing a proof-of-concept for the use of the Cell processor for Lattice QCD calculations. The creation of this code demanded substantial platform-specific knowledge to obtain good performance, but for applications built around a single kernel – as the lattice QCD code is around Dslash – the effort invested in producing a Cell-optimised version of the kernel can render an order-of-magnitude performance improvement compared with conventional platforms.
# Contents

1 Introduction ............................................. 1

2 The Cell processor ........................................ 3
   2.1 The Cell Architecture ...................................... 3
      2.1.1 System-wide features ................................. 3
      2.1.2 The SPE .............................................. 6
      2.1.3 DMA and the Master Flow Controller .................. 7
   2.2 Cell-based systems ....................................... 8
   2.3 Programming the Cell processor ......................... 9
      2.3.1 libspe2 ............................................... 9
      2.3.2 Accelerated Library Framework (ALF) ................. 11
      2.3.3 Software Cache Library (libcache) .................. 13
      2.3.4 RapidMind SDK ...................................... 14
      2.3.5 Further techniques .................................. 16

3 Lattice QCD ................................................ 17
   3.1 Lattice QCD libraries ...................................... 17
   3.2 The QMT multi-threading framework ....................... 18
   3.3 Building QDP++ for the Cell ............................... 19
   3.4 Expression evaluation in QDP++ ........................... 20

4 Parallelising linear algebra in QDP++ ..................... 22
   4.1 SPE Runtime Management Library (libspe2) ................ 24
   4.2 Accelerated Library Framework (ALF) ..................... 27
   4.3 Software Cache Library (libcache) ....................... 29
   4.4 RapidMind SDK .......................................... 30

5 Benchmarks ............................................... 31
   5.1 Tuning parameters ........................................ 32
      5.1.1 libspe2 transfer size ................................. 32
      5.1.2 ALF workblock size .................................. 33
      5.1.3 libcache cache parameters ........................... 37
   5.2 Scaling .................................................. 38
   5.3 Comparisons ............................................. 40
# Table of Contents

## 6 Code Complexity & Maintenance
- 6.1 Source code metrics ........................................... 43
- 6.2 Qualitative evaluation ........................................ 45

## 7 A high performance Wilson Dslash operator
- 7.1 Decomposition and Communication .......................... 49
- 7.2 SIMD optimisation ............................................. 51
- 7.3 Further serial optimisation ................................... 57
- 7.4 Padding color matrix arrays .................................. 57
- 7.5 Generalisation for $\text{isign}$ and $\text{cb}$ parameters .... 58
- 7.6 Performance ..................................................... 59
- 7.7 Future work ..................................................... 62

## 8 Discussion ..................................................... 64

## 9 Conclusions ................................................... 67

## Bibliography .................................................... 68

## A Definitions ................................................... 72

## B Building ......................................................... 73
- B.1 Cross-compiling libxml2 for Cell ............................ 73
- B.2 Cross-compiling QDP++ for Cell ............................. 73

## C Modifications and additions to QDP++ ........................ 75

## D Variation of vaxpby performance with transfer size in libspe2 implementation ......................................................... 78

## E Test Programs .................................................. 81
- E.1 residue.py validation script .................................. 81
- E.2 t_blas_cell ...................................................... 82
- E.3 t_dslashm_cell .................................................. 85
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Performance comparison of implementation techniques on the vaxpby operation running on 8 SPEs.</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Metrics for SPE code only</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>Metrics for PPE code only</td>
<td>45</td>
</tr>
<tr>
<td>7.1</td>
<td>Comparison of Cell Dslash with other architectures</td>
<td>61</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The Cell processor die. Note the eight synergistic processing elements (SPEs), the power processing element (PPE) on the left, and the PPE’s L2 cache in the top-left. (Reproduced courtesy of IBM)</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Diagram of the Cell Broadband Engine architecture</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>QDP++ in the dependency tree for the Chroma application code</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Effect of DMA transfer size on performance of libspe2 vaxpby3 operation for a $16^4$ lattice</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>Effect of workblock size on performance of the ALF vaxpby3 operation for a $4^4$ lattice</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>Effect of workblock size on performance of the ALF vaxpby3 operation for an $8^4$ lattice</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Effect of workblock size on performance of the ALF vaxpby3 operation for a $16^4$ lattice</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>Effect of workblock size on performance of the ALF vaxpby3 operation for a $32^4$ lattice</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>Variation of libcache performance with number of SPEs for various cache parameters</td>
<td>37</td>
</tr>
<tr>
<td>5.7</td>
<td>Scaling of the libspe2 vaxpby3 implementation</td>
<td>39</td>
</tr>
<tr>
<td>5.8</td>
<td>Scaling of the ALF vaxpby3 implementation</td>
<td>39</td>
</tr>
<tr>
<td>5.9</td>
<td>Scaling of the libcache vaxpby3 implementation (2kB cache lines)</td>
<td>40</td>
</tr>
<tr>
<td>5.10</td>
<td>Scaling of the RapidMind vaxpby3 implementation</td>
<td>41</td>
</tr>
<tr>
<td>5.11</td>
<td>Scaling of all vaxpby implementations for $L = 16$</td>
<td>42</td>
</tr>
<tr>
<td>5.12</td>
<td>Scaling of all vaxpby implementations for $L = 32$</td>
<td>42</td>
</tr>
<tr>
<td>7.1</td>
<td>The chunk of the lattice allocated to one SPE, with one of the space dimensions not drawn</td>
<td>50</td>
</tr>
<tr>
<td>7.2</td>
<td>Data dependencies in Psi arrays for the calculation of each index of Chi</td>
<td>52</td>
</tr>
<tr>
<td>7.3</td>
<td>asmVis displaying the SPE optimised spin projection function (the surrounding loop over sites is removed). The left-hand column of instructions shows the arithmetic pipeline, while that on the right shows the logic pipeline. “X” characters indicate work being done, while “</td>
<td>” characters indicate a pipeline stall.</td>
</tr>
<tr>
<td>7.4</td>
<td>Performance of Cell-optimised Wilson Dslash</td>
<td>60</td>
</tr>
</tbody>
</table>
# Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>libspe2 thread creation procedure</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Functions for MFC operations on the SPE</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>The prototype of the ALF accelerator-side computational kernel function</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>Initialising libcache</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>A simple RapidMind program adapted from the RapidMind Basic Tutorial</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>Example operation in an application using QDP++</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Example special-case function for evaluating a z=ax+by operation</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Cell interface for vaxpby</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>SPE vaxpby operation</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>The Job structure describing the parameters of a vaxpby operation</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>The entry point of the SPE program used with ALF</td>
<td>28</td>
</tr>
<tr>
<td>4.5</td>
<td>The AlfJob structure</td>
<td>28</td>
</tr>
<tr>
<td>4.6</td>
<td>The vaxpby calculation loop including libcache load/store operations</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>vaxpby RapidMind program</td>
<td>30</td>
</tr>
<tr>
<td>4.8</td>
<td>Invocation of vaxpby RapidMind program</td>
<td>30</td>
</tr>
<tr>
<td>7.1</td>
<td>Part of a QDP++ Dslash calculation</td>
<td>48</td>
</tr>
<tr>
<td>7.2</td>
<td>Dslash operations for all lattice directions</td>
<td>49</td>
</tr>
<tr>
<td>7.3</td>
<td>Example of SIMD intrinsics</td>
<td>52</td>
</tr>
<tr>
<td>7.4</td>
<td>Example of using spu_shuffle</td>
<td>53</td>
</tr>
<tr>
<td>7.5</td>
<td>QDP++ generic scalar spin projection function</td>
<td>54</td>
</tr>
<tr>
<td>7.6</td>
<td>Cell SIMD spin projection function</td>
<td>55</td>
</tr>
<tr>
<td>E.1</td>
<td>residue.py</td>
<td>81</td>
</tr>
<tr>
<td>E.2</td>
<td>t_blas_cell.cc</td>
<td>82</td>
</tr>
<tr>
<td>E.3</td>
<td>t_dslashm_cell.cc</td>
<td>85</td>
</tr>
</tbody>
</table>
Acknowledgements

Jon Hill and Adrian Jackson provided supervision for this project, supplying invaluable guidance in project management as well as technical assistance.

Chris Maynard and Bálint Joó provided very useful advice regarding QDP++ and Lattice QCD in general, and the efforts of all the QDP++ authors are appreciated for their work producing a well structured and documented package.

Thanks to Mike McNamee (IBM Edinburgh) for putting us in touch with Joachim Jordan (IBM Böblingen), who generously granted access to the Cell blades used for development and benchmarking.

I’m grateful to EPSRC for funding for my year in Edinburgh, and to EPCC for providing a good working environment with a bottomless coffee pot. My fellow students made my time here all the more enjoyable, Florian Scharinger in particular provided useful advice on the code complexity metrics used in this project.
Chapter 1

Introduction

The Cell processor is a heterogeneous multi-core processor,[1] originally designed for media applications but potentially applicable to traditional high performance computing applications such as computational science. In this project, the use of the Cell processor for such applications is investigated, using Lattice Quantum Chromodynamics (LQCD), a highly computationally demanding problem from particle physics,[2], as a test case.

Interest in the possibility of computational science on the Cell processor,[3, 4, 5] is driven by its exceptionally high peak floating point capability. In single precision, current generation Cell hardware has a peak of around 200GFlop/s per chip,[1], compared to the tens of GFlop/s found in the latest conventional processors. However, the exploitation of this potential is obstructed by difficulties in programming the Cell processor, and especially difficulties in porting existing codes to 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 cache-mediated access to the same main-memory address space, the Cell has a hierarchical memory architecture,[6], where some processing cores have a small private memory called a local store. Movement of data to and from local stores is performed in software, in contrast to the hardware caching engines found in conventional processors. This explicit control of data movement provides independence from potentially sub-optimal hardware caching algorithms,[7], at the cost of demanding a greater effort from the programmer.

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.[8] 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[9, 10]. This project is limited in scope to single-node programs, but demonstrating the per-Cell performance possible for LQCD is an important step toward further research into a many-node Cell-based system.

To investigate both the sustained performance attainable on the Cell processor, and the difficulties involved in programming it, elements of an existing LQCD library called
QDP++ are ported to the Cell using a variety of programming models (Chapters 2, 3, 4). This permits a comparison of both performance obtained (Chapter 5) and programming effort (Chapter 6) for each potential approach. Using lessons learned from this process, a high performance implementation of a key LQCD kernel is implemented to provide an idea of what sustained performance may be achieved for LQCD codes on the Cell (Chapter 7). Chapters 8 and 9 discuss the results obtained and draw conclusions relevant to the original hypothesis of the Cell processor having potential for running scientific codes such as those used in LQCD.
Chapter 2

The Cell processor

The Cell processor\(^1\) consists of one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs)\(^2\). The PPE is a conventional PowerPC processor, including a 512kB L2 cache and a VMX unit which provides SIMD operations. The SPEs are SIMD processors, optimised for floating point operations on 128b vectors. Each SPE has a 256kB local store (LS) and associated 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: as shown in Fig.\(^2.1\), 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 – this is a key feature of the Cell processor design.

2.1 The Cell Architecture

2.1.1 System-wide features

The logical structure of the Cell processor is shown in Figure\(^2.2\). Processing elements (PEs) are connected via the Element Interconnect Bus (EIB), which provides a peak bandwidth of 204.8GB/s. 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

---

1 The Cell processor is variously referred to as the 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”.

2 The SPE is defined to encompass a core, LS and DMA engine. The term Synergistic Processing Unit (SPU) is also used, to refer to only the core and LS. The distinction between SPE and SPU is rarely relevant, and SPE is used throughout this report. Analogously, one may refer to the PPE minus its DMA engine using the term PPU.
Figure 2.1: The Cell processor die. Note the eight synergistic processing elements (SPEs), the power processing element (PPE) on the left, and the PPE’s L2 cache in the top-left. (Reproduced courtesy of IBM)
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: transfers to an SPE LS do not go via the PPE cache, although 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 Fig. 2.2, 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. However, the limited I/O bandwidth (25.6GB/s shared between both IOIF) means that when using a dual-processor Cell system in this way, for best performance one must take into consideration the bandwidth bottleneck between PE on different physical processors. Either I/O interface may be connected to an I/O device, typically a “south bridge” (the chipset component usually leading to a PCI bus and disk controller). In the PlayStation 3, one I/O interface is dedicated to the graphics processor. In a dual-processor Cell blade, IOIF0 is used to connect both Cell processors together[12], and IOIF1 leads to a separate south bridge for each processor. If the Cell were adapted to be used in a massively parallel machine, one or both of the I/O interfaces could potentially act as a dedicated connection to a network interface. A Cell node with more than 2 processors is possible, but would require an extra component in the form of a switch to connect the EIBs of multiple processors together in a coherent manner: no such system has been implemented at this time.

Typically, an application runs its main thread of execution on the PPE, while exe-
cuting certain computationally intensive procedures on the SPEs. The PPE core is a conventional PowerPC processor with a 512kB L2 cache needing little further explanation here, while the SPE core is an original design with its own distinct instruction set. SPE instructions are mostly analogous to the standard PowerPC VMX vector instructions\[13\] found in the PPE, but not similar enough to map directly onto the VMX portion of the PowerPC instruction set.

2.1.2 The SPE

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 as an add or multiply instruction does, 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. Single precision floating point operations on the SPE are not fully IEEE-754 compliant\[6\], having no exceptions, and only supporting rounding towards zero. However, the double precision unit is IEEE-754 compliant. Performing double precision (DP) arithmetic on current-generation Cell hardware is much slower than working single precision, but this is set to change in future revisions of the platform: plans for a future Cell processor featuring 100GFlop/s peak DP performance have been announced\[9\].

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\(^3\) 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).

### 2.1.3 DMA and the Master Flow Controller

Each PE has a dedicated hardware DMA engine, known as a Master Flow Controller (MFC), which performs transfers asynchronously with respect to the program’s execution. For best performance and scalability, DMA operations should always be performed from the SPE MFCs rather than the PPE, mostly simply because this brings eight MFCs to bear on the task as opposed to one. Issuing MFC commands from the SPE program is an efficient process: the put and get functions provided in the SDK are implemented by simply writing their arguments to certain special registers. The overall cost of issuing an MFC command is around 20 cycles.

The bandwidth achieved by the MFC depends on the size of transfers performed. Kistler et al.\(^{16}\) used micro-benchmarks to determine that maximum bandwidth for get operations from main memory is achieved for transfers with size equal to or greater than 4kB, or 2kB for puts. This maximum is imposed by the bandwidth of the EIB port, 25.6GB/s. Combining this limit with the peak floating point capacity per SPE, 25.6GFlop/s, the MFC can only transfer in one single precision floating point number for every four floating point operations that the SPE core can perform. This constrains on attempts to reach peak performance: a floating point number must take part in at least 4 operations for every time it is transferred in and out. In practice, this means that programs should load data on to SPEs for repeated operations where possible, rather than purely streaming it.

Since the MFC performs transfers asynchronously, it is possible to overlap computation and communication. This permits hiding the cost of communications using double buffering, wherein one set of data is transferred while another is worked on. Whether double buffering is possible or worthwhile depends on the data dependencies of the task, as well as whether there is sufficient data to permit splitting the work into buffers without requiring transfers so small as to be inefficient.

As well as contiguous transfers, the MFC supports “DMA lists”, where a list of addresses and sizes is composed in the LS, and this structure’s location is passed to the MFC command. Using DMA lists one may perform scatter/gather operations, although the aforementioned conditions of size and alignment apply to each element in the list. Since it is the global addresses and not the LS addresses that are stored in the list, such a scatter operation scatters a contiguous region in LS to disparate locations in main

---

\(^3\)The total floating point performance is 256GFlop/s if the PPE’s VMX unit is included, but it is not here since we do not consider programming approaches which include calculations on the PPE.
memory, while a gather operation gathers disparate locations in main memory to a contiguous region in LS. DMA lists also have a lower issue overhead than many small contiguous DMA transfers, and thus may improve performance even when the data is contiguous in main memory.

Further to DMA operations, each MFC also provides so-called “signal” and “mailbox” communications, which transfer single 32b values. According to the Cell programmer’s handbook[6] section 3.2.2, these are intended for convenient management of SPE tasks. Mailbox messages are written to a fixed-length inbound queue on the receiving processing element or to an outbound queue on the sending processing element: each SPE has two outbound queues and one inbound queue. Each SPE has two signal channels, each holding a single message without queueing, where which channel to write to is controlled by the sender. Signals and mailboxes are accessible using straightforward APIs on the PPE and the SPE.

2.2 Cell-based systems

Cell-based systems have been manufactured in several form factors, including the PlayStation 3[17], a PCI-E accelerator board[18], and a blade server. The current IBM Cell blade offering, the BladeCenter QS20 Server[12], is a dual-Cell NUMA system, with 1GB of RAM in total. The system runs a modified version of the Fedora Core Linux distribution. Each blade has dual gigabit Ethernet controllers, and the option of 1 or 2 InfiniBand 4x interfaces.

Current BladeCenter solutions based on double-width dual-processor blades accommodate 84 Cell processors in a 42U rack. With each processor having a peak floating point capacity of 200GFlop/s in single precision, one rack has an aggregate peak of 16.8TFlop/s. A BlueGene/L rack has a peak performance of 5.73TFlop/s[19]. As such, a Cell BladeCenter solution offers comparable computational density to a massively parallel architecture such as BlueGene/L, although the current Cell processor only obtains this performance in single precision.

Future Cell-based BladeCenter products announced by IBM include single-width blades featuring a double precision revision of the Cell processor[20][21], providing an even greater computational density. It is anticipated[9] that the double precision Cell processor will have a 100GFlop/s peak, providing the same 16.8TFlop/s per rack as the current cell products, but in double precision. This remains competitive with the BlueGene/P[10], which provides 14.2TFlop/s per rack, establishing the Cell architecture’s potential as an HPC platform.
2.3 Programming the Cell processor

A software development kit (SDK) for the Cell is publicly available, consisting of packages from IBM, as well as components produced by the Barcelona Supercomputing Centre (BSC). Two compilers are included, the open source GNU compiler collection (GCC) which was adapted for the Cell by BSC, and IBM’s proprietary compiler (XLC). XLC tends to provide superior automatic SIMD-isation of code, although when code is hand-optimised for the SPE this becomes less significant. The XLF Fortran compiler is also included, but this only outputs code for the PPE, whereas GCC and XLC are provided both for the PPE and the SPE.

In addition to the compilation toolchain, the SDK includes many libraries. These include libraries for parallelising work across SPEs, which are described in the following sections, as well as mathematics libraries for the SPE for tasks including Fourier transforms, image processing and matrix manipulation. Importantly, the SDK also includes extensive documentation [6, 22, 23, 14], information from which is used throughout this report.

The SDK may either be installed natively on a PowerPC or Cell machine, or as a cross-compiler on an x86 machine. If Cell hardware is unavailable, programs may be tested using the included full system simulator. This provides an accurate model of a dual-processor Cell system, including various features for performance analysis. The speed of the simulator compared to a real system is sufficiently low that it is suitable only for running small benchmarks.

In this project, four approaches to programming the Cell are considered in detail: low-level programming (section 2.3.1 libspe2), a workblock model (section 2.3.2 ALF), an SPE software cache (section 2.3.3 libcache), and a high-level data parallel abstraction (section 2.3.4 RapidMind). The libraries and techniques used to achieved these approaches are described in detail in the following sections, before being applied in chapter 4.

In addition to the official Cell SDK, there are a number of third party packages providing higher level abstractions including both commercial products and research projects. Some of these are Cell specific [24], while some provide general multi-core mappings which can be run on a graphics card or conventional multi-core CPU as well as on the Cell [25, 26].

2.3.1 libspe2

The SPE management library libspe2 [27], included in the Cell SDK, provides low-level functionality for loading code onto an SPE and executing it. All synchronisation and data movement is left up to the programmer. This low level approach provides great flexibility, but also demands relatively complex code to manage the SPEs, as well as complex code in the SPE programs to perform DMA operations and synchronisation.
as well as the actual task underway. Note that although other methods may use the libspe2 library, from this point onwards the term libspe2 is used in this report to refer to programs that do not use any higher level abstractions.

Creating a SPE thread using libspe2 requires only a few straightforward function calls. Firstly, a “context” is created (spe_context_create), where a context represents one physical SPE. Next, a SPE binary is loaded (spe_program_load) to the context, and finally the SPE program is run (spe_context_run). This procedure is summarised in Listing 2.1 for some imaginary SPE program which is at spe_program_ptr. This pointer would be resolved at link time to a separately compiled SPE program, embedded in the PPE executable as data. Arguments to the functions are explained in detail in the libspe2 documentation[27].

Listing 2.1: libspe2 thread creation procedure

```
// Create an SPE context, which maps to one physical SPE
spe_context_ptr_t context = spe_context_create (SPE_MAP_PS, NULL);

// Load the SPE program image from spe_program_ptr into that SPE
spe_program_load (context, &spe_program_ptr);

// A parameter which must be passed as a pointer
unsigned int entry = SPE_DEFAULT_ENTRY;

// Run the context, blocking until SPE program terminates
spe_context_run (context, &entry, 0, job, NULL, NULL);
```

The spe_context_run function blocks until the SPE program terminates, so a separate PPE thread is required for each SPE thread. This may be achieved using the well-known POSIX threads library[28]. On the SPE side, the program must issue DMA commands to load input data to the LS and return results to main memory (or to another SPE in case of a pipelining parallelisation). Convenience wrappers for issuing DMA commands are provided in the SDK, for example those shown in Listing 2.2. Note that these are not part of libspe2, which is only responsible for managing the SPE threads, but are nevertheless included in this section since they are only needed when no higher level framework is in use.

Listing 2.2: Functions for MFC operations on the SPE

```
(void) mfc_put (volatile void *ls, uint64_t ea, uint32_t size, 
  uint32_t tag, uint32_t tid, uint32_t rid)

(void) mfc_get (volatile void *ls, uint64_t ea, uint32_t size, 
  uint32_t tag, uint32_t tid, uint32_t rid)
```

MFC functions are named from the point of view of the SPE, such that to put is to write to main memory from LS and to get is to read from main memory to LS. The ls pointer holds a 32b address in the LS, while ea holds a 64b “effective address” (EA) specifying a location either on-chip or in main memory. Both the EA and the LS address must be 128B-aligned. The transfer size (size) may only be 1, 2, 4, 8, 16, or a multiple of 16. The transfer size must be no greater than 16kB. These limitations are
inherent to the hardware, so any transfers not meeting these constraints require software work-arounds. Particularly, transfers greater than 16kB may be accomplished using a simple wrapper:

```c
void big_get (volatile void *ls, uint64_t ea, uint32_t size, uint32_t tag, uint32_t tid, uint32_t rid)
{
    int blocksize = 16 * 1024;
    int blocks = size / blocksize;
    int remainder = size % blocksize;

    for (int i = 0; i < blocks; ++i) {
        mfc_get (ls + i * blocksize, ea + i * blocksize, blocksize, tag, tid, rid);
    }
    if (remainder > 0)
        mfc_get (ls + blocks * blocksize, ea + blocks * blocksize, remainder, tag, tid, rid);
}
```

### 2.3.2 Accelerated Library Framework (ALF)

The accelerated library framework (ALF)\[29\] provides a “work block” (WB) model for single program multiple data operations, where the main program runs on a “host” processor and WBs are executed on “accelerator” processors. ALF is designed such that the “host” processor on which the main program runs is not necessarily the PPU of the Cell providing the “accelerators” (SPEs). For example, the forthcoming RoadRunner machine is to be an x86/Cell hybrid\[8\], so the host may be an x86 processor, while the accelerators may be provided by a Cell processor. On the Cell, ALF is implemented on top of libspe2, but this is entirely hidden to the application developer.

To implement a calculation using ALF, a computational kernel is written for the accelerator, which work on one WB at a time, where a WB is exposed as input and output buffers. The host program defines WB objects that specify regions of the input and output datasets to work on (i.e. what will go in the input/output buffers), and then passes these WB objects to the ALF runtime for execution on the accelerators. The accelerator computational kernel is implemented by the application developer as a function called `alf_comp_kernel`, the prototype of which is shown in Listing 2.3.

Listing 2.3: The prototype of the ALF accelerator-side computational kernel function

```c
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 );
```
The input and output buffer addresses are stored in `p_input_buffer` and
`p_output_buffer` respectively. `p_parm_ctx_buffer` is a pointer to arbitrary
user data which is transferred along with the input and output buffers, which might in-
clude information such as the size of the input and output buffers. `p_task_context`
is a persistent buffer associated with each accelerator context (i.e. an SPE) – this could
be used for calculations involving accumulation over multiple WBs. `current_count`
and `total_count` are not used unless the WB has the multi-use attribute set, in
which case the WB is executed more than once sequentially and these parameters sup-
ply progress information.

In a minimal ALF program, the `alf_comp_kernel` function is the only accelerator-
side code that the application developer must write. However, greater flexibility in
communications is possible if the programmer elects to create DMA lists for the input
and output buffers by hand on the accelerator (“accelerator-side partitioning”) rather
than having ALF generate them implicitly on the host (“host-side partitioning”). This
prevents host-side generation of transfer lists from becoming a scaling bottleneck, but
more importantly allows arbitrary data movement, such as overlapping the input and
output buffers to decrease the overall LS space required and thus allow greater WB
sizes. Further, when using accelerator-side partitioning one can create “multi-use” WBs
which allow substantial reduction in overhead by having ALF invoke the same WB
repeatedly with a sequentially increasing integer argument, such that the accelerator-
side transfer code can select a different region of memory to operate upon depending
on this argument.

Where possible, ALF overlaps DMA transfers with computation using its inbuilt double-
buffering capability. Rather than allowing applications to specify when double buffer-
ing should be used, it is automatically enabled when twice sum of the input, output
and overlap buffers does not exceed 240kB. The 240kB figure is an approximate upper
bound on the amount of free space which may be available after the program and stack
are allocated: approaching this limit with the data size risks overwriting stack or pro-
gram data. The application programmer provides an estimate of the stack size required
by the SPE program when initialising the task object: this does not need to be exact,
but should be updated if the SPE code pushes anything large onto the stack in order to
allow ALF to safely make use of available local store.

To provide more than one operation via ALF within one program, multiple “tasks”
are created, each associated with a different SPE kernel. In the absence of context-
switching on SPEs, each SPE can only be associated with one task at a time, i.e. only
a single kernel program is loaded on an SPE at a time. This creates a problem if one
wishes to implement a program which uses multiple small ALF kernels, since in be-
tween calls to different operations the relevant kernels must be transferred to the SPEs.
One way around this would be to implement a monolithic SPE kernel which takes a
parameter specifying which operation to perform, although this would destroy the ab-
straction of the kernel in the SPE program from the control flow in the main program.
In the programs presented here, the situation of having multiple ALF tasks is simply
avoided.
2.3.3 Software Cache Library (libcache)

An alternative to explicitly transferring data in and out of the local store is to use a software caching algorithm. This approach is particularly suited to dealing with unpredictable access patterns over datasets larger than the local store. As such, it is not an especially good fit for functions like vaxpby, iterating once over a dataset in a predictable way. It is included in this study since it is a potentially useful approach for other operations, and using it in this context provides an opportunity to measure the overheads involved.

Software caching is implemented in the libcache library\[30\] included with the Cell SDK. Since it is implemented at the library rather than compiler level, it is necessary to include explicit calls to the library. In the implementation of libcache used, these calls resolve first to preprocessor definitions, and then to \texttt{inline} C functions. On a cache read operation, libcache performs the following procedure:

1. Determine which cache line the value resides in
2. Determine the value’s offset within the cache line
3. Check if the line is in LS, and return its address in LS
4. If the line was not in local store, perform a DMA get to obtain it, wait for completion and then return its address in LS

Although these operations are implemented quite efficiently, they nevertheless constitute a significant overhead if each loaded value is involved in only a few floating point operations.

The software cache is configured at compile time, using parameters including the size of a cache line, the number of sets, and the associativity of the cache. These parameters may be tuned for best performance, although since they are set at compile time this cannot be done “on the fly”. The reason that these parameters are set at compile time is that libcache makes extensive use of preprocessor macros to reduce runtime overheads. Parameters are set using preprocessor definitions before the inclusion of the libcache header, as shown in Listing 2.4.
Once the cache is established, it is accessed using the `cache_rd` and `cache_wr` functions. If a requested value is not in cache, `cache_rd` blocks until it is loaded from main memory. To avoid this, non-blocking functions are provided which may be used to improve performance at the cost of code complexity. The non-blocking libcachef interface is not explained here, since it is not used in the performance tests presented later in this report.

Apart from being used directly via libcachef, software caching can be part of an OpenMP implementation for the Cell, as reported by Eichenberger et al. [31]. In this case, superior performance to hand-inserted load and store calls may obtained as the compiler would have some awareness of loop structure and thus be able to insert appropriate prefetches to avoid cache misses on sequentially accessed data.

### 2.3.4 RapidMind SDK

The RapidMind SDK [32] provides a high level toolkit for multi-core development. The programmer accesses RapidMind functionality via a C++ library, expressing data-parallel operations on RapidMind datatypes. It includes “backends” for the Cell processor, graphics cards, and conventional SMP multi-core CPUs. The RapidMind SDK is a commerical product, used here under a developer evaluation license.

Operations to be distributed across cores are expressed as “RapidMind Programs”, which are C++ objects defining a series of operations. RapidMind programs are analogous to functions: they have well defined inputs, outputs and scope. Data to be processed in parallel by the RapidMind runtime is stored using RapidMind types, including array objects which do not store data themselves, rather pointing to existing data. Using arrays such as these, it is possible to encapsulate the RapidMind-based portions of a program rather than using RapidMind types throughout.

A simple RapidMind example program from the RapidMind Basic Tutorial [33] is shown
in Listing 2.5. Note that while the function takes inputs of type Value3f, it is invoked with arguments of type Array<1,Value3f> (a one dimensional array of 3-vectors of floats). In this way, the RapidMind::Program object is a so-called “stream program”, where the operation specified on the scalar type is automatically performed on all elements of the array in parallel.

Listing 2.5: A simple RapidMind program adapted from the RapidMind Basic Tutorial

```cpp
#include <rapidmind/platform.hpp>
using namespace rapidmind;

int main()
{
    // General initialization of the platform
    rapidmind::init();

    const int num_elements = 10000;

    // Create arrays to pass input data to the stream program
    Array<1,Value3f> input1(num_elements);
    Array<1,Value3f> input2(num_elements);

    // Create an array to receive the output from the program
    Array<1,Value3f> output;

    // The stream program that will be executed on the data
    Program prg = RM_BEGIN {
        In<Value3f> a; // first input
        In<Value3f> b; // second input
        Out<Value3f> c; // output
        c = a + b; // operation on the data
    } RM_END;

    // Execute the stream program (and send its output to the output array)
    output = prg(input1, input2);
}
```

Note that the source of the RapidMind program is included inline with the main program. This is significant for Cell development, since it eliminates the need for separate SPE and PPE source code, making source code more readily understood as functionally related subprograms may be in the same source file rather than being divided into PPE and SPE code. Additionally this makes compilation simpler since only one compiler is needed.

A RapidMind program is compiled when it is first called, to machine code for which ever backend is in use (Cell, GPU, etc). The overhead of runtime compilation is negligible when a program is used many times, although when benchmarking care must be taken not to include the very first invocation of the program in timings. Runtime compilation adds the convenient ability to run the same binary on conventional multi-core CPUs as well as the Cell processor. The ability to rapidly re-target development to different platforms is an advantage of the RapidMind model. However, one is restricted to
the use of RapidMind’s inbuilt compiler, where the optimisation provided by compilers from the platform vendor might be superior.

2.3.5 Further techniques

There have been many other approaches to the question of adapting code for the Cell processor. A review including most of them is available by Buttari et al. Some of these are available publicly, while some are proprietary, and some are research projects while others are commercial products. These are not explained in detail here, but are listed for the interest and to illustrate the diversity of the emerging field of Cell/multi-core software development tools.

Peakstream is a proprietary Cell/GPU/Multi-core environment similar to RapidMind. Sequoia is a custom language designed for programming hierarchical memory systems such as the Cell. The Offload API uses a workblock approach similar to ALF, and has been used successfully to accelerate a computational chemistry code. An Interface Description Language (IDL) framework for RPC on the Cell is supplied with the SDK, but is deprecated in favour of ALF. IBM have published impressive performance results from an OpenMP compiler known as the Octopiler, but this technology remains unavailable. IBM research also implemented a system called MPI Microtask, where SPU programs are written using MPI-like functions for communications – this also has not been released. Barcelona Supercomputing Center have developed an annotation-based single-source system called Cell Superscalar (CellSs), which provides pragmas to specify functions which are candidates for offload, similar to OpenMP but without explicitly specifying parallel regions. CorePy is a Python-based environment in which low-level assembly operations are exposed to the Python programmer such that Python can be used to write the entire program without having to link in C or assembly, without sacrificing low-level optimisation. During this project, an additional Cell programming environment called Gedae became available, mentioned here only in the interests of completeness.
Chapter 3

Lattice QCD

Quantum Chromodynamics (QCD) is a theory of the strong interaction which binds together hadrons, such as the neutrons and protons in an atomic nucleus. Lattice QCD (LQCD) is a formulation of QCD on a four dimensional space-time lattice, which approaches physical correctness as the lattice spacing approaches zero. LQCD is used to perform numerical simulations\(^2\) investigating so-called “non-perturbative” phenomena, that is those which cannot be investigated using analytical methods. LQCD simulations are extremely computationally demanding, to such a degree that massively parallel machines have been custom-built solely to run LQCD calculations. The most recent of these is the QCD On a Chip (QCDOC) machine\(^{41, 42}\). QCDOC achieves very high computational density using a custom-designed low-power processor, and high sustained performance using a low latency on-chip interconnect. The custom nature of the machine permits a good balance of communications bandwidth, memory bandwidth and floating point processing power for the application. The QCDOC design predates and has much in common with the more general purpose BlueGene/L and its successor BlueGene/P, illustrating the role of LQCD as a driver of HPC technology.

3.1 Lattice QCD libraries

A number of mature libraries designed for LQCD calculations are available. Many are targeted at one specific platform or another, such as the Columbia Physics System (CPS)\(^{43}\) which was originally designed for QCDSP (a forerunner of QCDOC), and later adapted to QCDOC. The package used in this project is the more platform-neutral QCD Data Parallel (QDP++) library, developed relatively recently under the auspices of the United States SciDAC program\(^{11}\) for scientific software development. QDP++ is used mainly by the Chroma application code, for which it provides lattice-wide classes whose contents are automatically distributed across a parallel machine. The version 1.23.2 tarball\(^1\) of QDP++ was used: although development of QDP++ is ongoing in a

\(^1\)Available at time of writing from http://usqcd.jlab.org/usqcd-docs/qdp++/
public CVS repository, the option of tracking CVS HEAD was rejected due to the issues associated with working on a changing codebase.

There are a number of SciDAC QCD packages, organised loosely into “levels”\[\text{11}\]. Level 1 packages include a linear algebra library (QLA) and a message passing library (QMP), which are used by the C data parallel library (QDP/C), itself a level 2 package. The QDP I/O library (QIO) is also considered a level 2 package. Level 3 includes highly optimised modules for key calculations which applications may use alongside QDP, while level 4 is reserved for various high level components such as reusable data analysis and visualisation tools. Application codes such as Chroma are built on top of these components, such that application authors may concentrate on science without being distracted by parallelisation and implementing common linear algebra operations. QDP++ does not fit neatly into this scheme of levels, since it includes all required linear algebra operations rather than depending on QLA. However, QDP++ does rely on QMP and QIO. As such, the actual stack used in a Chroma application is simply that shown in Fig. \[\text{3.1}\].

![QDP++ in the dependency tree for the Chroma application code.](image)

Some of the fast assembly kernels used in existing QCD code are generated by a tool called Bagel\[\text{44}\], which generates such code for certain algorithms on QCDOC and BlueGene/L architectures. Although it may be possible to extend Bagel to target the SPEs of the Cell processor, this approach is rejected on the basis that extending Bagel in this way would be a time-consuming project in itself.

### 3.2 The QMT multi-threading framework

As multi-core architectures become the norm\[\text{45}\], particularly on the popular Opteron-based Cray XT3/4 machines, there is increasing interest in enabling QCD codes to operate effectively on such systems through multi-threaded execution. One approach
to this in to the Chroma/QDP++ suite is known as QCD Multi-Threading (QMT)[46]. According to the README file, “QMT is a software library providing OpenMP like fork-join multi-thread APIs. The current implementation works on i386 and x86-64 using pthread and other optimizations.”

If QMT could be adapted to the Cell, this would be a great benefit to the applications programmer, since the same interface would provide Cell multi-threading as well as traditional SMP multi-threading. However, the current pthreads-based QMT does not appear to be general enough for this: invoking a parallel region in QMT’s fork-join model is done by calling an existing function via the `qmt_call` function:

```c
typedef void (*qmt_userfunc_t) (void *usrarg, int thid);

int qmt_call (qmt_userfunc_t func, unsigned int count, void* arg);
```

This mechanism presents two main problems in migrating the workload to the SPE. Firstly, functions are identified by their address, which not the same on the SPE as it is on the PPE, since code for each is compiled separately. Secondly, one does not know in advance which functions are to be called using QMT, as such one does not know which code to load on the SPE, and the 256kB local store is insufficient to load all possible QDP++ operations. Due to these issues, QMT is not considered further in this report – it is explained here only to illustrate an existing SMP multi-core mechanism and the difficulty in mapping SMP parallel code onto the Cell.

### 3.3 Building QDP++ for the Cell

The QDP++ source tree uses the GNU autotools[47] (autoconf, automake etc) to generate Makefiles. Various compile time options are set using arguments to the `configure` script, including architecture-specific options such as `--enable-sse` as well as simulation parameters such as `--enable-Nc` (set the number of colours). In this project, simulation parameters are left as their default values, and inter-node communications are disabled (`--enable-parallel-arch=scalar`). Full details of the build process are given in section B.2. QDP++ also depends on libxml2 for manipulating its XML file formats, and this must also be built for Cell despite not using the XML capabilities in this project. Details of the libxml2 build process are given in section B.1.
To accommodate Cell-specific extensions in the code which runs on the PPE, additional source files are simply added to Makefile.am files in the relevant directory. SPE programs are compiled separately, outside the autotools setup. This has the disadvantage of requiring two invocations of make to compile the whole program, but the effort of neatly integrating the SPE compilation process with its own compilers and options would not be worthwhile given that this project’s code consists of benchmarks and proofs-of-concept code rather than proposed additions to QDP++.

In a further departure from the autotools style, multiple variants of the Cell code co-exist in the same source files, varying regions delimited with preprocessor conditional blocks. Which Cell code to build is specified using preprocessor definitions in a header file (cell-config.h), rather than with configure parameters. A neat mechanism to select between different Cell implementations is not a requirement since a production code would use only one chosen implementation. Also, this approach renders a more rapid turnaround in testing different implementations, since a full re-run of configure is not required.

More detail on the way that Cell-specific modifications are accommodated in the QDP++ source is provided in appendix C.

3.4 Expression evaluation in QDP++

The QDP++ library uses C++ templates to present application developers an intuitive API providing common arithmetic operations acting on various datatypes across multiple lattice sites. For example, Listing 3.1 shows how an application can elegantly express operations upon a LatticeFermion object, in this case a “scalar × vector + scalar × vector” operation (the LatticeFermion class represents the fermion field as vectors of complex numbers).

Listing 3.1: Example operation in an application using QDP++

```
LatticeFermion x, y, z;
Real a, b;

z = a * x + b * y;
```

While providing a neat API, the object oriented design of QDP++ imposes limitations on how operations may be parallelised across SPEs from a main program running on the PPE. EIB transfers operate on the byte level, meaning that lattice data in the form of C++ objects cannot readily be migrated from PPE to SPE. It is not only the lattice data that is stored in this form, arithmetic operations themselves are represented as objects. The C++ RunTime Type Information (RTTI) extension cannot solve this due to the differences in name-mangling with the PPE and SPE toolchains – in order to extract type information in a meaningful way it would be necessary to convert the C++ type of an object into some identifier, such as an enum. As well as introducing overhead, this
would impose an undesirable maintenance cost, as the Cell-specific type identifier code
would have to be kept up to date with any changes to the classes used in the code as a whole.

Expressions involving QDP++ datatypes are resolved to specific (potentially optimised)
functions at compile time by the included Portable Expression Template Engine (PETE)[48].
PETE provides a capability to parse expressions and resolve them to either generic func-
tions which iterate across sites performing the operation at each site, or to special-case
implementations which treat the operation as a whole. Since this is done at compile
time using C++ templates, it does not impose a runtime overhead. For example, the op-
eration expressed in Listing 3.1 is resolved by PETE to the function prototype in Listing

Listing 3.2: Example special-case function for evaluating a z=ax+by operation.

```c++
// From qdp++/include/scalarsite_generic/...
qdp_scalarsite_generic_blas.h

template<>
void evaluate( OLattice< TVec > &d,
    const OpAssign &op,
    const QDPExpr<
        BinaryNode<OpAdd,
            Reference< QDPType< TScal, OScalar< TScal >> >,
            Reference< QDPType< TVec, OLattice< TVec >> > >,
            Reference< QDPType< TScal, OScalar< TScal >> > >,
            BinaryNode<OpMultiply,
        Reference< QDPType< TScal, OScalar< TScal >> >,
        Reference< QDPType< TVec, OLattice< TVec >> > >,
        OLattice< TVec > &rhs,
    const Subset& s);,
```

The implementation of such an `evaluate()` function may obtain pointers to the raw
floating point data from the operand objects and thus call a corresponding C function
operating on arrays of floats. The implementation of such a function on the Cell pro-
cessor is explored in Chapter 4.
Chapter 4

Parallelising linear algebra in QDP++

Once operations on QDP++ types have been resolved to evaluate() calls as described in section 3.4, one may ignore the object-oriented design of QDP++, and work directly on arrays of floating point numbers. The floating point numbers are referred to as REAL, which is a QDP++ type definition set to either float or double at compile time. To allow for more specific discussion, a representative operation is chosen: \( z = ax + by \) (scalar \( a \) and \( b \), vector \( x \), \( y \) and \( z \)), also known as vaxpby. In the existing platform-independent (“generic”) implementation, this resolves to the function called vaxpby3, which takes \( a, b, x \) and \( y \) as input and outputs to \( z \). The generic vaxpby3 simply iterates over all three arrays calculating \( ax + by \) at each index. In each of the Cell implementations presented in this chapter, this function is replaced with a version which shares the work between the SPEs. In a multi-node system, this division of lattice data between SPEs would be an extra layer beneath the division of lattice data amongst nodes.

To compare the programming methods under consideration, the vaxpby operation is implemented four times for the four methods: libspe2, ALF, libcache and RapidMind. The multiple implementations in the same source code are accommodated using preprocessor conditionals. All four versions expose a common interface to the application, summarised in Listing 4.1.
Listing 4.1: Cell interface for vaxpby

```cpp
namespace Cell {

  /*
   * init_num_spus: number of SPUs to try and use
   * init_blocksize: blocksize parameter (ignored by some impl.)
   */
  void initialize (int const init_num_spus, int const init_blocksize);

  void finalize ();

  void vaxpby3 (REAL *z, REAL a, REAL *x, REAL b, REAL *y, int n_3vec);
}
```

A call to `Cell::initialize` is appended to the `QDP_initialize` function in `qdp_scalar_init.cc`, and a call to `Cell::finalize` is prepended to the `QDP_finalize` function in the same file. The call to the generic `vaxpby3` in `qdp_scalarsite_generic_blas.h` is replaced with a call to `Cell::vaxpby3`. The final change to the existing code is due to the requirement that Cell DMA operations operate on 128B aligned addresses. This cannot be efficiently encapsulated in the Cell-specific code, and is achieved by changing the definition `QDP_ALIGNMENT_SIZE` in `qdp.h` from 16 to 128. All further Cell code is restricted to Cell-specific source files, avoiding unnecessary changes to existing QDP++ code.

Apart from RapidMind, each technique needs a compute kernel written for the SPE. For vaxpby, this is quite simple to implement using SPE SIMD intrinsics, as shown in Listing 4.2. A detailed explanation of this kind of SIMD programming is given in section 7.2, wherein a more complex case is considered. This kind of explicit SIMD programming is necessary to obtain good performance on the SPE: the version shown in Listing 4.2 runs in 5% of the time taken by the output of `spuxlc -O3 -qhot` on the equivalent scalar loop, as measured using the Cell simulator.
Listing 4.2: SPE vaxpby operation

```c
void cell_vaxpby3(REAL *zp, REAL *ap ,REAL *xp, REAL *bp, REAL *yp, int n_3vec)
{
    register vector REAL a = spu_splats (*ap);
    register vector REAL b = spu_splats (*bp);
    vector REAL *x = (vector REAL *) xp;
    vector REAL *y = (vector REAL *) yp;
    vector REAL *z = (vector REAL *) zp;
    register vector REAL x_tmp, y_tmp, z_tmp;

    for (i = 0; i < (n_3vec * 6) / 4; ++i) {
        x_tmp = x[i];
        z_tmp = spu_mul (a, x_tmp);
        y_tmp = y[i];
        z_tmp = spu_madd (b, y_tmp, z_tmp);
        z[i] = z_tmp;
    }
}
```

All code written and benchmarks performed in this report use single precision floating point arithmetic. This is because the current Cell processor’s double precision arithmetic is very slow compared to single precision, and thus working in double precision would lead to such slow serial code that the efficiency of communications and parallelisation in general would be more difficult to measure in benchmarks. Also, working in single precision should provide a good impression of double precision performance on future revisions of the Cell processor which are planned to have much improved double precision performance[9].

Subsequent sections in this chapter describe implementations of vaxpby3 using the various techniques to be evaluated: libspe2 (section 4.1), ALF (section 4.2), libcache (section 4.3) and RapidMind (section 4.4).

4.1 SPE Runtime Management Library (libspe2)

libspe2 was introduced in section 2.3.1. Provided with only a minimal thread-management framework, the programmer may use arbitrary operations, including inter-SPE communications. However, uniform-cost operations on continuous data such as vaxpby require only a simple set of operations: to transfer a block of input data to each SPE, and to transfer results from each SPE to main memory once the calculation is complete. Thus, the program must only accomplish the following operations as simply and efficiently as possible:

- Divide the input data between SPEs and transfer it to the appropriate local stores
• Perform the vaxpby operation in parallel on all SPEs
• Collect the results back into main memory from the SPE local stores

A fork/join style is rejected, since the overheads associated with a pthreads fork/join, let alone the SPE thread startup, are much too high to incur at every call to a parallel linear algebra routine. Rather, the SPE threads are initialised (by corresponding PPE threads) when QDP++ is initialised, and terminated when the QDP++ finalize function is run.

As such, a mechanism is required by which the SPE threads may idle until being “woken up” by the main program running on the PPE. There are a number of possible ways to do this:

• Mailbox communications, provided in libspe2[27] on the PPE side and as language extensions[15] on the SPE side
• Atomic operations, provided by libsync[30]
• Condition variables, also provided by libsync

The simplest approach would be to use mailbox communications, and have SPE programs poll their inbound mailbox while idle, waiting for the PPE to write a message. However, it is found that mailbox communications do not provide sufficient performance[1]. What is needed is mechanism which can synchronise up to 16 threads in a time which is negligible compared to the runtime of the communications and computation.

Using atomic operations for synchronisation is accomplished quite simply, by having a status variable (an integer) in main memory for each SPE thread, which it polls using atomic_read until the PPE thread uses atomic_set to write a “magic” value which causes the SPE execution to advance. Although at first it seems that this mechanism requires a round-trip to main memory, in fact it can remain on-chip since each MFC caches several 128 byte blocks (see Section 6.2.1 in [6]), and atomic read/write operations can work with these cached values. To make best use of these caches, the integers are spaced out in main memory such that they lie on 128 byte boundaries: each one is associated with a different cache line.

Experiment showed that the time required to synchronise 8 threads is around 20 times less when using atomic operations than when using mailboxes. A detailed analysis of this is not presented here, since the synchronisation mechanism is only required to be “good enough” for the task at hand, i.e. its runtime is small compared to that of the calculation. Atomic operations were used in preference to condition variables in the interests of simplicity, since condition variables require both a condition variable object and a mutex, whereas synchronisation using atomic operations uses only a single integer’s effective address for each SPE thread.

---

1Results indicating the performance of mailbox communications are not presented in this report since mailboxes are not used in any of the final code
Each SPE program is passed a pointer to a structure of type `Job` (Listing 4.3) as an argument when it is initialised using the `spe_context_run` function. `Job` specifies the addresses of the input arrays, as well as the number of items to be operated on: `Job.n_3vec` complex 3-vectors (6 floats). `Job` also specifies the parameter `blocksize`, which is the size of DMA transfer used in the procedure specified below.

Listing 4.3: The `Job` structure describing the parameters of a vaxpby operation.

```c
struct Job {
    int rank;
    int n_3vec;
    int blocksize;
    REAL a;
    REAL b;
    unsigned long long x_ptr; // 64bit effective address
    unsigned long long y_ptr;
    unsigned long long z_ptr;
} __attribute__((aligned(128)));
```

The SPE program allocates input and output buffers \((x, y, z)_{0,1}\), which are two sets of buffers to allow double buffering. Subject to the synchronisation routine described above, when an SPE finds its status variable to be nonzero it transfers its `Job` structure into local store and proceeds with the following procedure:

- Transfer in \((mfc\_get)\) the first `blocksize` bytes of the `x` and `y` arrays to local `x_0` and `y_0` buffers
- Wait on the initial `x` and `y` transfers
- Calculate the total bytes to be transferred in each array:
  \[\text{arraysize} = \text{Job.n}_3\text{vec}\times6\times\text{sizeof(REAL)}\]
- For \(i=0\) to `arraysize/blocksize`:
  - Initiate DMA transfer of “block \(i+1\)” parts of `x` and `y` into local `x_{(i+1)}%2` and `y_{(i+1)}%2`
  - Perform computation on local buffers `x_{i%2}` and `y_{i%2}`, storing result in `z_{i%2}`
  - Initiate DMA transfer \((mfc\_put)\) of `z_{i%2}` to corresponding location in `z`
  - Wait for transfer of block \(i+1\)
- Perform a cleanup step if `arraysize % blocksize != 0`
- Wait for all DMA transfers to main memory to complete
- Set \((\text{atomic\_set})\) the status variable to 0
Meanwhile, once the PPE thread has set all the SPEs status variables to nonzero values, it enters a loop over all SPEs and for each one loops on its status variable being nonzero (while(atomic_read(...))). Once this loop completes, it is guaranteed that every SPE has completed its work and transferred its results to main memory.

The simple block decomposition used here is justified on the basis that the workload is uniform throughout the dataset, and the simplicity of the SPEs (particularly that they run no O/S) makes their timing very dependable, such that they will complete the same work in the same time. For a workload which was not so readily load balanced, a more complex parallelisation might be required, involving a locked queue of tasks, or a block cyclic distribution of work.

Note that all the DMA operations are initiated from the SPE program, while the PPE is responsible only for causing the SPEs to initialise and then waiting on their completion via their status variables. This is in line with the good practise of performing as many operations as possible SPE-side to prevent the PPE becoming a bottleneck when using many SPEs. A peer-to-peer SPE synchronisation mechanism could also be of interest for even greater efficiency, but was not investigated here since the synchronisation mechanism used was found to constitute only a negligible proportion of the runtime of this program.

4.2 Accelerated Library Framework (ALF)

Introduced in section 2.3.2 ALF provides much of the task management that was implemented by hand in the libspe2 version. All that is required on the PPE side is to break the operand arrays into workblocks (WBs), dispatch these WBs to ALF and await their completion. The SPE code may consist of nothing more than a kernel (alf_comp_kernel) which ALF calls having handled communications itself (“host-side transfer lists”), or the programmer may implement their own communications routines for greater flexibility (“accelerator-side transfer lists”).

The alf_comp_kernel implementation used for the vaxpby operation is shown in Listing 4.4: the input and output buffers correspond to $x$, $y$ and $z$, while parm_ctx_buffer holds an AlfJob structure (Listing 4.5) containing $a$, $b$ and the number of items to compute.
Listing 4.4: The entry point of the SPE program used with ALF

```c
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)
{
    AlfJob *jobinfo = (AlfJob*) p_parm_ctx_buffer;

    REAL *x = (REAL*) p_input_buffer;
    REAL *y = (REAL*) p_input_buffer;
    y += jobinfo->n_3vec * 6;
    REAL *z = (REAL*) p_output_buffer;

    cell_vaxpby3(z, &(jobinfo->a) ,x, &(jobinfo->b), y, ~
                 jobinfo->n_3vec);

    return 0;
}
```

Listing 4.5: The AlfJob structure

```c
struct AlfJob {
    int n_3vec;
    REAL a;
    REAL b;
};
```

During initialization on the PPE side, an alf_task_info object is created specifying the SPE kernel to be used, input and output buffer sizes, and parameters relating to the generation of data transfer lists. Thereafter, at each call to vaxpby, the dataset is split up into blocks of an arbitrary length and for each of these a WB is initialised (alf_wb_create), and enqueued (alf_wb_enqueue) for execution. The size of a WB is to be tuned for best performance (section 5.1.2). Once all workblocks have been enqueued, a barrier synchronisation point is associated with the alf_task_info object (alf_wb_sync), such that execution of the PPE program does not proceed until all workblocks have been completed (alf_wb_sync_wait).

In cases where the input data may be destroyed during the calculation, ALF provides a capability for overlapping the input buffer with the output buffer, as explained in section 2.3.2. When performing accelerator-side partitioning, this overlapping may be done to reduce the total local store space required to process a WB. This allows the use of greater WB sizes than would otherwise be possible, increasing the allowable range of the WB size parameter when tuning for performance. The other possibility afforded by accelerator-side partitioning is multi-use workblocks. These are not used here, despite a strong potential for improved performance, because when using multi-use workblocks...
ALF becomes more of a convenient utility library than a programming model: one simply creates as many workblocks as SPEs and has programs the data transfers for each one explicitly in the SPE code. Using this approach the communication patterns should be almost identical to the libspe2 version, albeit accomplished using slightly less code.

### 4.3 Software Cache Library (libcache)

To some extent, libcache allows a more traditional SMP style of programming on the SPE. However, it has one key deficiency in this respect: a lack of coherency between the software caches of the SPEs. For this reason, it is necessary to manually flush the cache of a SPE when there is any chance of another SPE accessing a line that the first may have in a dirty state. Even when the mutex functionality provided by libsync are used, this requirement for manual flushing of the cache remains. Due to this caveat, libcache-based code does not provide a clean abstraction of the LS. Also, as stated in section 2.3.3 libcache not well suited to predictable-access functions such as vaxpby. For these reasons, the inclusion of libcache in this project serves more to provide an evaluation of the overheads associated with libcache then it does to demonstrate libcache “in its element” or to propose it as a truly high-level approach.

The code implementing the vaxpby operation using libcache has a great deal in common with the libspe2 version, since the division of work and the synchronisation of SPE programs is the same. The key difference between the pure libspe2 version and the libcache version is the replacement of explicit DMA operations before and after the calculation with libcache calls within the calculation loop. Thus, the inner calculation loop is changed to the form shown in Listing 4.6.

#### Listing 4.6: The vaxpby calculation loop including libcache load/store operations

```c
// Where CACHE_NAME was 'blas' at initialisation
#define LOAD(ea) cache_rd (blas, (unsigned)(ea))
#define STORE(ea, val) cache_wr (blas, (unsigned)(ea), (val))

for (int i = 0; i < (n_3vec * 6) / 4; ++i) {
    x_tmp = LOAD(&x[i]);
    z_tmp = spu_mul (a, x_tmp);
    y_tmp = LOAD(&y[i]);
    z_tmp = spu_madd (b, y_tmp, z_tmp);
    STORE (&z[i], z_tmp);
}
```

---

2Libcache could not possibly be doing the communications required for such coherency since it is not supplied with addresses to other SPEs in use, nor to the address space of the PPE thread.
4.4 RapidMind SDK

To perform the vaxpby operation using the RapidMind SDK, it must be expressed as a RapidMind stream program. The RapidMind program used is shown in Listing 4.7.

Listing 4.7: vaxpby RapidMind program

```cpp
rapidmind::Program vaxpby3_prog = RM_BEGIN {
    using namespace rapidmind;
    In<Value4f> x;
    In<Value4f> y;
    Out<Value4f> z;
    z = rm_a * x + rm_b * y;
} RM_END;
```

The variables rm_a and rm_b are defined in the enclosing namespace with type rapidmind::Value1f. The program works identically if the In and Out variables hold Value1f instead of Value4f, but benchmarks during development indicated that using Value4f provided better performance. This is probably related to the fact that a Value4f corresponds to a 128b SPE register.

To invoke this program, RapidMind types referring to the operand data and result are constructed, and then the program is called as if it were a C++ function, as shown in Listing 4.8.

Listing 4.8: Invocation of vaxpby RapidMind program

```cpp
void vaxpby3 (REAL *z, REAL a, REAL *x, REAL b, REAL *y, int n_3vec) {
    using namespace rapidmind;
    const unsigned int n_float_vecs = n_3vec * 6 / 4;
    Array<1, Value4f> x_arr;
    Array<1, Value4f> y_arr;
    Array<1, Value4f> z_arr;

    rm_a = a;
    rm_b = b;

    access_array(x, host::Value1ui(n_float_vecs), x_arr);
    access_array(y, host::Value1ui(n_float_vecs), y_arr);
    access_array(z, host::Value1ui(n_float_vecs), z_arr);

    z_arr = vaxpby3_prog (x_arr, y_arr);
}
```

When initialising RapidMind, one does not specify the exact number of SPEs to use, rather one sets a parameter called max_spus which determines the maximum number of SPEs to use. The RapidMind runtime uses a varying number of SPEs depending on the size of operation performed, by an undocumented algorithm[49]. The use of less than max_spus in some cases may be confirmed by running a RapidMind program inside the Cell simulator and observing the GUI indicator of which SPEs are active.
Chapter 5

Benchmarks

All benchmarks were run on a QS20 Cell blade as described in section 2.2. The blade is a relatively noisy benchmarking environment, since there is a full Linux operating system running on it including many daemon processes which may be migrated between the two processors at the discretion of the scheduler. Anecdotal experience suggested that longer benchmark durations (upwards of 20 seconds) tended to produce higher performance results: this could indicate that the scheduler takes some time to migrate system daemons away from the processor on which the benchmark is running. It seems a reasonable assumption that the scheduler never migrates the benchmark program’s main thread to the PPE on the other Cell from that containing the SPEs being used for computation.

Each timing result is an average over 3 runs of around 20 seconds each. The results displayed here are given in Flop/s, which is obtained from the timing by the formula

\[
\text{Flop/s} = \frac{72 \times L^4 \times N_{\text{iterations}}}{t}
\]

where \(N_{\text{iterations}}\) is the number of times the procedure was repeated, \(L\) is the side length of the 4D lattice and \(t\) is the wall clock time taken. 72 is the number of floating point operations per lattice site\(^1\). Dataset sizes used in tests are stated in lattice size rather than number of floating point numbers, since this is the context in which the vaxpby operation is used in QDP++ – vaxpby is invoked by multiplying two QDP::LatticeFermion objects by scalars and adding the results. This procedure is carried out by the simple test program t_blas_cell.cc, included as appendix E.2.

One could record the time/site/iteration instead of the Flop/s: one is trivially converted to the other. Flop/s is presented here because it allows rapid comparison of the performance achieved with the peak performance of the processor, as well as with the performance of the procedure on other architectures.

\(^1\)Two multiplies and an add, for each of the \(4 \times 3 \times 2\) floating point numbers in a LatticeFermion at each site. \(4 \times 3 \times 2 \times 3 = 72\)
Results from these calculations were verified by comparing the result array with that generated by compiling the test program against an unmodified QDP++ library compiled and run on the Cell. The comparison is performed using a simple python script shown in appendix E.1: the deviation of the specimen result from the original result is calculated as the root mean square of the differences between corresponding elements of the output arrays from each program. A tolerance is applied to this result, such that values of the RMS difference up to $10^{-5}$ are accepted as a “pass”. Small variations are expected for two reasons. Firstly, the program is performing single precision calculations and the SP units of the SPEs are not IEEE754 compliant, particularly with respect to rounding. Secondly, a fused multiply-add operation is used in some cases, which is liable to produce a different result than a separate multiply and add, since it rounds only once at the end of the calculation, in contrast to the two rounding operations when using separate multiply and add instructions.

Benchmarks are performed on 1, 2, 4, 8 and 16 SPEs, where the number used is denoted by $N_{spus}$. Varying $N_{spus}$ rather than considering only the 8 SPEs on one Cell processor is done for several reasons. Firstly, it provides an indication of the effectiveness of the parallelisation strategy used. Secondly, using 16 SPEs from one PPE program on a dual-processor machine may be preferable to running an 8-SPE instance of the program on each processor.

The remainder of this chapter is divided into 3 sections: firstly, those implementations having tunable parameters are tested for various values of the parameters in order to determine the optimal configuration. Secondly, all implementations are benchmarked on various numbers of SPEs to produce scaling results to evaluate the efficiency of their parallelisation. Finally, selected results from each implementation are presented side-by-side to give a comparison of the performance achieved by each programming technique.

5.1 Tuning parameters

5.1.1 libspe2 transfer size

In the libspe2 implementation, each SPE is assigned a $1/N_{spus}$ fraction of the dataset to process. As explained in section 4.1, the SPE program transfers and processes blocks of its workload in a streaming fashion. The size of these blocks is arbitrary, up to the 16kB hardware limit on DMA transfers. Note that this transfer size is defined per-array, such that a 1kB blocksize corresponds to 1kB of $x$, 1kB of $y$ and 1kB of $z$, or a total of 2kB input and 1kB output.

Fig. 5.1 shows the effect of the block size (and thus size of DMA transfer) on the performance of the vaxpby operation for a lattice of size $L = 16$. Corresponding benchmarks for different lattice sizes may be found in Appendix D, although they do not display any significant qualitative differences in the dependence on transfer size.
Performance initially increases with transfer size, levelling off at a around 4kB. This result agrees with Kistler et al. [16]. This behaviour is intrinsic to the hardware: small block sizes incur latency and management overheads which degrade their bandwidth. Beyond around 4kB, these overheads are not measurable and peak bandwidth is achieved. Performance increases with $N_{\text{spus}}$ up to 8, falling slightly for 16. The dependence on $N_{\text{spus}}$ is explored in more detail in section 5.2.

The performance achieved in the limit of a large transfer size is 5.4GFlop/s. Recall that vaxpby is the $z = ax + by$ operation: for each vector index $i$ the program performs three floating point operations (one multiply and one add), two loads ($x_i$ and $y_i$) and one store ($z_i$). Thus, for 4B floating point storage, each Flop/s requires 4B/s of bandwidth: a 5.4GFlop/s execution is using 21.6GB/s of bandwidth. This is of similar magnitude to the peak main memory bandwidth, 25.6GB/s. It is therefore reasonable to conclude that the main bottleneck experienced by the libspe2 vaxpby is the main memory bandwidth.

### 5.1.2 ALF workblock size

In the ALF implementation, the work is divided into work blocks (WB) of arbitrary size. Because of the way input and output buffers are overlapped, the total buffer memory required in the LS is equal to twice the allocation from each array per WB, i.e. a WB taking 1kB each from $x$, $y$ and $z$ would required 2kB in the LS for buffers. The LS space required per WB is used as the parameter in the following benchmarks. Figs. 5.2, 5.3, 5.4, and 5.5 show the variation of performance with WB size, for lattice sizes...
Figure 5.2: Effect of workblock size on performance of the ALF vaxpby3 operation for a 4^4 lattice

$L = 4, L = 8, L = 16$ and $L = 32$ respectively.

Many of these graphs feature a jump at a total buffer size of around 120kB. This is due to how ALF handles double buffering: it is automatically enabled when two sets of buffers would not require more than 240kB\cite{29}, i.e. when the buffer storage requirement for each workblock is not more than 120kB.

The dependence on WB size is dramatically different for different lattice sizes. The $L = 4$ case (Fig. 5.2) shows similar behaviour for all values of $N_{spus}$: two plateaus divided at the double buffering threshold (120kB), where performance is markedly greater beyond this threshold. A $4^4$ lattice corresponds to $x, y$ and $z$ arrays 73kB long, so for most WB sizes considered only a few WB are created: the overhead of initialising the procedure appears to be dominant in this case, since there is little variation with number of SPEs or WB size.

As with $L = 4$, in the $L = 8$ results (Fig. 5.3), performance increases dramatically when double buffering is disabled (i.e. for total buffer sizes greater than 120kB). This is also a result of the limited size of the dataset: an $8^4$ lattice corresponds to $x, y$ and $z$ arrays 400kB long, which for a total buffer size of 120kB gives only 7 WB. When several SPEs are used, each one gets only a few WB to process, and thus insufficient serial work to overcome startup overheads. However, this does not explain why double buffering causes such a degradation in performance rather than simply not helping. It is possible that the ALF queueing algorithm is failing to provide load balance in the case of few work blocks and double buffering: if WB are assigned to processors in twos then such a granularity would not be sufficient to provide load balance for a few tens of
Figure 5.3: Effect of workblock size on performance of the ALF vaxpby3 operation for an $8^4$ lattice

Figure 5.4: Effect of workblock size on performance of the ALF vaxpby3 operation for a $16^4$ lattice
For the larger $L = 16$ and $L = 32$ lattices (Figs. 5.4 and 5.5), the behaviour is more complex. Here, the performance decreases at the double buffering threshold for up to 4 SPEs, but increases or remains constant for 8 or 16. Also, those series showing a decrease at the double buffering threshold tend to have plateaued prior to this. This indicates that in these cases the programs run efficiently enough that double buffering is beneficial to their performance, i.e. these lattice sizes lie outside the regime of having too few WB for these numbers of SPEs. For the 8 and 16 SPE cases, the performance is still as good or better when double buffering is disabled (above the threshold), indicating that there is still not enough work to go around when using this many SPEs. The 8 SPE case in particular shows a marked increase in performance at the threshold for the $L = 16$ lattice, but is almost flat at the threshold for the $L = 32$ lattice where the dataset is larger.

In the $L = 32$ plot (Fig. 5.5), the 8 and 16 SPE series converge to around 5GFlop/s, but the 8 SPE series reaches this limit for much smaller workblocks. This could indicate that the performance hit from contention for the PPE workblock queue is significant; whether this explanation makes sense depends on whether the workblocks are shared out by SPEs accessing a master queue or whether workblocks are assigned asynchronously to SPE queues by the PPE. Determining this would require analysis of a

\[ \text{Figure 5.5: Effect of workblock size on performance of the ALF vaxpby3 operation for a } 32^4 \text{ lattice} \]

\[ \text{WB. The queueing algorithm is not documented, so to determine it would require close inspection of the library source code which is beyond the scope of this project.} \]

\[ \text{For the larger } L = 16 \text{ and } L = 32 \text{ lattices (Figs. 5.4 and 5.5), the behaviour is more complex. Here, the performance decreases at the double buffering threshold for up to 4 SPEs, but increases or remains constant for 8 or 16. Also, those series showing a decrease at the double buffering threshold tend to have plateaued prior to this. This indicates that in these cases the programs run efficiently enough that double buffering is beneficial to their performance, i.e. these lattice sizes lie outside the regime of having too few WB for these numbers of SPEs. For the 8 and 16 SPE cases, the performance is still as good or better when double buffering is disabled (above the threshold), indicating that there is still not enough work to go around when using this many SPEs. The 8 SPE case in particular shows a marked increase in performance at the threshold for the } L = 16 \text{ lattice, but is almost flat at the threshold for the } L = 32 \text{ lattice where the dataset is larger.} \]

\[ \text{In the } L = 32 \text{ plot (Fig. 5.5), the 8 and 16 SPE series converge to around 5GFlop/s, but the 8 SPE series reaches this limit for much smaller workblocks. This could indicate that the performance hit from contention for the PPE workblock queue is significant; whether this explanation makes sense depends on whether the workblocks are shared out by SPEs accessing a master queue or whether workblocks are assigned asynchronously to SPE queues by the PPE. Determining this would require analysis of a} \]

\[ \text{2The 8 SPE series on the } L = 32 \text{ lattice decreases in performance very slightly at the threshold, this is attributable to benchmark noise} \]
5.1.3 libcache cache parameters

Fig. 5.6 shows the variation of performance with cache parameters for a fixed lattice size of $16^4$. Each of the configurations shows good scaling, but only up to 1 or 2 GFlop/s, around a third of the best libspe2 and ALF results. This near-linear scaling is a result of low serial performance on the SPEs, resulting from the explicit load and store calls which have a significant overhead as they are implemented in software by libcache. Additionally, cache misses cause blocking communications, however even the non-double-buffered ALF case outperformed this implementation, indicating that cache misses are not the primary source of degraded performance here.

One would expect that when using a cache to access contiguous data linearly, longer cache lines would result in fewer misses and thus higher performance. This is borne out by the results in Fig. 5.6: series are ordered by cache line length with the greatest being given by the 2kB cache line case. This ordering is maintained for $4^4$, $8^4$ and $32^4$ systems, for which benchmarks were run but are not shown here.

The variation in the number of sets used was done simply to accommodate larger cache lines: 64 sets with 1024kB cache lines would exceed the LS capacity. The simplicity of the data access pattern in vaxpby means that the number of sets and the associativity of the cache are unlikely in themselves to have a measurable effect on performance.
5.2 Scaling

The effectiveness of each implementation’s parallelisation is evaluated by plotting scaling graphs for each implementation. Each series is a strong scaling result, with multiple dataset sizes shown on each graph to indicate how much data an implementation requires to operate efficiently. The ALF and libspe2 results are selected from the parameter optimisation benchmarks by simply taking the highest performing transfer size or workblock size for each system size and number of SPEs. The libcache results are taken from the highest-performing configuration (2kB cache lines). The RapidMind implementation did not have any tunable parameters, so the results shown in this section represent all RapidMind benchmarks run.

Fig. 5.7 shows the scaling of the libspe2 implementation. All system sizes perform best on 4 SPEs. One might attempt to overcome this by running two processes per Cell (taking advantage of the multi-threading capability of the PPE), but this would not be successful for those cases already obtaining around 5-6GFlop/s, since in these cases main memory bandwidth is the limiting factor. The peak of each series at 4 SPEs may be explained as a trade-off between computational power of multiple SPEs and the increased contention for the main memory interface associated with greater numbers of SPEs.

The $4^4$ system shows disappointing performance, while the $8^4$ system runs at up to 5GFlop/s, close to the 5.4GFlop/s maximum of the $16^4$ and $32^4$ systems. For up to 8 SPEs, the $16^4$ system gives performance identical to the $32^4$ system: both are limited by memory bandwidth. On 16 SPEs, the $16^4$ system shows degraded performance compared to the $32^4$, showing that the $16^4$ system provides enough data to obtain bandwidth-limited performance on 8 SPEs but not on 16, implying that to within a factor of two the $16^4$ system represents the lower bound on the system size to obtain bandwidth-limited performance.

Fig. 5.8 shows the scaling of the ALF implementation. The $32^4$ system outperforms the $16^4$ system, indicating that even the relatively large $16^4$ sub-lattice is not large enough to reach the large-system performance limit: the ALF implementation does not scale as well as the libspe2 implementation.

The $4^4$ system shows very poor performance which barely varies with the number of SPEs, indicating that runtime is dominated by the startup cost of creating workblocks on the PPE. The $8^4$ system does better, reaching a maximum of around 3GFlop/s on 8 SPEs, although the libspe2 implementation ran at 5GFlop/s on 4 SPEs. The $16^4$ and $32^4$ systems both reach plateaus from 8 to 16 SPEs, indicating that it is the startup overhead rather than a per-SPE cost that is making the $16^4$ system under-perform with respect to the $32^4$ system. However, the $32^4$ system features a peak for 4 SPEs, showing that the per-SPE overhead becomes measurable in the >5GFlop/s regime.

Fig. 5.9 shows the scaling of the libcache implementation. For the larger systems, this shows a near-linear scaling, although this is symptomatic of poor serial performance rather than a good parallelisation: the libcache code uses the same thread-management
Figure 5.7: Scaling of the **libspe2** vaxpby3 implementation

Figure 5.8: Scaling of the **ALF** vaxpby3 implementation
mechanisms as the libspe2 version. As for the other implementations, the $4^4$ system shows notably poorer performance, as each SPE is assigned only a few kB of the dataset.

Fig. 5.10 shows the scaling of the RapidMind implementation. This displays a great sensitivity to system size: the $4^4$ and $8^4$ systems show poor performance, the $8^4$ system only scaling to 2 SPEs. The $16^4$ system scales to 6 SPEs, although only performs at just over 3GFlop/s, while the $32^4$ system achieves over 5GFlop/s from 6 SPEs. This performance is competitive with the libspe2 and ALF implementations.

**5.3 Comparisons**

Figs. 5.11 and 5.12 compare the scaling of each implementation for systems of size $16^4$ and $32^4$ respectively.

For the smaller $16^4$ system, libspe2 is the clear leader in performance, although it reaches its maximum at only 4 SPEs. ALF has a maximum performance around 10% less than libspe2, and reaches it at 8 SPEs. RapidMind does less well in terms of maximum performance, peaking at 6 SPEs and becoming slower for larger numbers of SPEs. libcache has almost linear scaling, but remains slower than all other implementations for all numbers of SPEs, due to the low performance of its SPE code.

The larger $32^4$ system, having 16 times more sites, closes the gaps between libspe2, ALF and RapidMind. Each of these now reaches maximum performance at 4-6 SPEs, with maxima between 5GFlop/s and 5.5GFlops. Considering the ease of programming
of the RapidMind implementation, it is impressive that it achieves this performance. However, if these results were not approaching memory bandwidth limits then the libspe2 – and possible ALF – implementation would have more scope to exceed the performance of the RapidMind version. libcaché still lags behind with just over a third of the performance of the other implementations.

Table 5.1 compares the performance of each implementation for 8 SPEs (1 Cell processor), including the runtime per site as well as the floating point performance. These are simply the same data plotted in the previous figures, in textual form for clarity.

![Figure 5.10: Scaling of the RapidMind vaxpby3 implementation](image)

Table 5.1: Performance comparison of implementation techniques on the vaxpby operation running on 8 SPEs.

<table>
<thead>
<tr>
<th></th>
<th>Libspe2</th>
<th>ALF</th>
<th>Libcache</th>
<th>RapidMind</th>
</tr>
</thead>
<tbody>
<tr>
<td>MFlop/s</td>
<td>5393</td>
<td>4642</td>
<td>1077</td>
<td>3265</td>
</tr>
<tr>
<td>ns/site/iteration</td>
<td>13.4</td>
<td>15.5</td>
<td>66.8</td>
<td>22.1</td>
</tr>
</tbody>
</table>

Wall-time in nanoseconds (ns): lower is better. MFlop/s: higher is better.
Figure 5.11: Scaling of all vaxpby implementations for $L = 16$

Figure 5.12: Scaling of all vaxpby implementations for $L = 32$
Chapter 6

Code Complexity & Maintenance

In this chapter, the source code used to obtain the performance results in the previous chapter is evaluated in the context of code quality and development effort. Code quality could be considered to fall into the category of development effort, since it is highly influential of the maintenance effort required. Both quantitative metrics and qualitative discussion are included.

6.1 Source code metrics

Quantitative code complexity metrics are measured using a tool called cccc\[50\], version 3.pre84. This program outputs a variety of statistics, but here only two are selected: the number of lines of code (LOC) and McCabe Cyclomatic Complexity\[51\] (MCC). The LOC count is calculated excluding whitespace and newlines and thus is a count of the number of statements. The MCC is a standard code complexity metric based on analysis of the control flow graph of a subprogram, counting the number of independent possible paths of execution. According to the cccc documentation, “In the case of C++, the [MCC] count is incremented for each of the following tokens: ‘if’, ‘while’, ‘for’, ‘switch’, ‘break’, ‘&&’, ‘||’”. The whole-program MCC is calculated by simply adding the MCC calculated for each function.

No code complexity metric is perfect\[52\], and LOC and MCC are relatively blunt instruments. Recent research focusing on inter-procedural analysis\[53\] and object oriented characteristics\[54\] has provided metrics which are better correlated with real-world data on maintenance work, but these are inapplicable to this code for two reasons. Firstly, the Cell-specific extensions to QDP++ are not written in an OO fashion. Secondly, the global procedure of executing the calculation spans processing elements, and thus is not confined to one program: as such automated inter-procedural metric calculations fail as they are unaware of inter-process communications. A parallel-aware metric would be desirable, but unfortunately no such metric is widely accepted, let alone available for the Cell programming environment.
Table 6.1 shows the results of running cccc on the SPE program of each implementation. For the RapidMind implementation, which does not include SPE code per se, the code constituting the RapidMind Program object is used for metric calculations. Two sets of figures are presented for ALF, one for the case of generating DMA lists implicitly on the host, and one where transfer lists are generated explicitly in the SPE code to allow buffer overlapping.

The SPE program of the libspe2 version has almost twice as many LOC as the next-longest version, and a 5 times the MCC. The libspe2 and ALF versions share the same vaxpby routine, which is a 4x hand-unrolled version of that shown in chapter 4, itself accounting for 43 LOC and having a MCC of 1. The large MCC result for the libspe2 version is a result of features such as checking pointers before freeing them, loops over the number of buffers used, and series of “else if” constructs conditional on the read from the atomic status variable used during synchronisation with the SPE.

The difference between the host-side partitioning and accelerator-side partitioning ALF results illustrates the development cost of implementing accelerator-side partitioning. Recall that accelerator-side partitioning is necessary to support the largest workblock sizes, which in some cases gave the best performance (section 5.1.2).

The libcache version shares much of the control complexity of the libspe2 version, albeit without the explicit management of SPE-side buffers. Also, the vaxpby kernel used in the libcache version is not hand-unrolled since the overhead of cache loads and stores lessens the usefulness of such unrolling, further decreasing the LOC. This makes the LOC comparison between the host-side partitioning ALF version and the libcache version slightly artificial since without the hand unrolling the host-side partitioning ALF version would be the shorter.

The RapidMind version is only 5 LOC, consisting of a few declarations and a single line performing the calculation. All of the SPE thread management and communications is performed by the RapidMind runtime.

Table 6.2 shows the equivalent metrics for the PPE code. The first three implementations show little variation, since they all do essentially the same thing: call some initialisation functions, then divide the dataset into chunks and assign these to SPEs (or in the case of ALF enqueue them) each time vaxpby is called. ALF has a slightly greater LOC due to the verbose code required to initialise a task object in initialize. RapidMind is notably different in that no PPE-side code is required to divide up the dataset, since
Table 6.2: Metrics for PPE code only

<table>
<thead>
<tr>
<th>Implementation</th>
<th>LOC</th>
<th>MCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>libspe2</td>
<td>120</td>
<td>7</td>
</tr>
<tr>
<td>ALF</td>
<td>133</td>
<td>9</td>
</tr>
<tr>
<td>libcache</td>
<td>115</td>
<td>7</td>
</tr>
<tr>
<td>RapidMind</td>
<td>54</td>
<td>2</td>
</tr>
</tbody>
</table>

this is handled entirely by the RapidMind runtime. This leads to substantial reductions in LOC and particularly MCC, since there are no loops over either the dataset or the SPEs.

Overall, we have the unsurprising result that the low-level libspe2 implementation (which obtained the best performance) has long and complex code, while the high-level RapidMind code (which performed less well) has very short and simple code. The other two implementations lie in between, although on the SPE side they are closer to the RapidMind implementation in terms of cyclomatic complexity than to the libspe2 implementation.

6.2 Qualitative evaluation

In terms of programmer-effort required for a working program, RapidMind is undoubtedly far ahead of the rest of the methods used: a working prototype of the vaxpby operation was created in less than an hour, and for large datasets performs as well as the libspe2 or ALF versions. In contrast, the libspe2 version took several days to get working reliably, due in part to the explicit address calculations and the relatively complex logic required to implement double buffering. The ALF version, despite having simpler code, took comparable effort to the libspe2 version when accelerator-side partitioning was used, since all the transfer operations still had to be worked out, while at the same time being aware of the layout of the ALF input and output buffers to perform overlapping correctly. The libcache version reused most of the code from the libspe2 version, apart from replacing the explicit buffer transfers with software cache loads and stores. This was fairly straightforward to do, but renders poor performance even for large numbers of SPEs on large datasets, as seen in chapter[5].

The synchronisation mechanism used in both the libspe2 and libcache versions took some trial and error to perfect, after some failed attempts to implement a fast synchronisation using mailboxes. However, once the use of atomic operations was established, the code was straightforward to implement thanks to the libsync library.

Beyond the programming itself, the libspe2, ALF and libcache implementations turned out to be quite sensitive to their parameters (transfer size, workblock size, cache parameters), requiring an extra tuning step after programming. This was not difficult for the libcache (larger cache lines are better) and libspe2 (transfers should be larger than
4kB) cases. However, the dependence of ALF performance on the workblock size was very complex, depending on the number of SPEs used and the system size. More predictable ALF performance could have been obtained using multi-use workblocks, but as explained in section 4.2 this option was rejected on the basis that it destroyed the ALF programming model and essentially replicates the libspe2 program.

The conclusion supports the source code metrics, in that the subjective opinion of the developer is that as one would expect the RapidMind implementation was by far the most straightforward, while libspe2 the most involved. However, the difficulty of programming and tuning the ALF implementation came as a surprise, particularly given that it still did not perform as well as the libspe2 version.
Chapter 7

A high performance Wilson Dslash operator

The previous chapters identified problem areas in attempting to parallelise linear algebra on the Cell. The most important factor constraining performance of the best performing technique (libspe2) was found to be the bandwidth to main memory. Rapidmind required a large lattice size to run well, while libcache-based communications were found to severely limit performance, not even approaching the memory bandwidth limit. ALF delivered comparable performance to the libspe2 version, but had a complex dependence on the size of workblock used, which may require tuning for each operation implemented. In response to these issues, this chapter presents an attempt at maximum performance of a QCD kernel on the Cell processor, paying no regard to issues of readability or maintainability. A different approach is used here than in previous chapters. Rather than attempting to encapsulate a single linear algebra operation, a series of inter-related operations are combined, such that the data transfer happens once at the beginning and end of the process, rather than in between each step. This is intended to give the SPE program more floating point operations to perform per data transferred. This code is implemented using libspe2 in preference to anything higher level, in the interests of maximum control to avoid any avoiding idiosyncrasies of other frameworks.

The most computationally intensive part of lattice QCD calculations is inverting the so-called Dirac operator\cite{19}, an operation referred to as “Dslash”. In this chapter, a special-case Dslash routine is presented which operates only in single precision, only on 4 dimensional systems, and only when the lattice data is laid out in a red-black checkerboard (so-called rb2 layout). The exact routine under used here is the “Wilson Dslash”, which is commonly used as a benchmark for lattice QCD codes\cite{55,19}. In this project, the physical or mathematical meaning of these operations is not required: all that is done is to directly transform an existing scalar Dslash procedure into a Cell-optimised form.

An example reference implementation of the Wilson Dslash using the QDP++ data
parallel framework is included in the examples/ subdirectory of the QDP++ source. Operations between neighbouring lattice sites are represented by operations between relatively shifted lattices, as illustrated in the last 3 lines of Listing [7.1], which shows a small part the QDP++ example Dslash implementation.

Listing 7.1: Part of a QDP++ Dslash calculation

```c++
/*
 * In actuality these are passed in as function arguments
*/
LatticeFermion& chi;
const multild<LatticeColorMatrix>& u;
const LatticeFermion& psi;
int cb;
int const otherCB = !cb;

/*
 * Temporary lattice-wide arrays of half-spinors
*/
LatticeHalfFermion tmp;
LatticeHalfFermion tmp2;
tmp[rb[otherCB]] = spinProjectDir0Plus(psi);
tmp2[rb[cb]] = shift(tmp, FORWARD, 0);
chi[rb[cb]] = spinReconstructDir0Plus(u[0]*tmp2);
```

The Dslash procedure takes as input a LatticeFermion object \(\psi\) (a spinor, which is 4 complex 3-vectors, for each site), and four LatticeColorMatrix (a colour matrix, which is a \(3 \times 3\) complex array, for each site) objects \(u\), one for each dimension. The output is the LatticeFermion object \(\chi\). The temporary LatticeHalfFermion objects store 2 complex 3-vectors for each site, representing the “projected” form of the data stored in a LatticeFermion.

The lattice-wide types (those with names starting Lattice) implement the \([\cdot]\) operator, with a Subset object as the argument. In this case, the subset is one of the red-black checkerboards, represented in the globally accessible constant array \(rb\), where \(rb[0]\) and \(rb[1]\) are subsets for each half of the checkerboard. When the LHS of an assignment statement specifies a lattice subset, the statement is executed only on sites in that subset.

The last 3 lines of Listing [7.1] – where the calculation is performed – perform a spin projection on the otherCB checkerboard of \(\psi\), cyclic-shift\[1\] the result in dimension 0 to overlap with the cb checkerboard of \(\chi\), and then multiply each element by \(u[d]\) (where \(d\) is one of the 4 axes) before performing a spin reconstruction and storing the result in \(\chi\). In a multi-node machine, the shift operations would imply communications with nearest neighbours, but in the single-node case they are essentially just index transformations.

\[1\]Lattice QCD simulations employ periodic boundary conditions
The full Dsslash operation consists of such operations for each lattice direction, as shown in Listing 7.2:

### Listing 7.2: Dslash operations for all lattice directions

```plaintext
1. \( \text{tmp[rb[otherCB]]} = \text{spinProjectDir0Minus(psi)}; \)
2. \( \text{tmp2[rb[cb]]} = \text{shift(tmp, FORWARD, 0)}; \)
3. \( \text{chi[rb[cb]]} = \text{spinReconstructDir0Minus(u[0]*tmp2)}; \)

4. \( \text{tmp[rb[otherCB]]} = \text{spinProjectDir1Minus(psi)}; \)
5. \( \text{tmp2[rb[cb]]} = \text{shift(tmp, FORWARD, 1)}; \)
6. \( \text{chi[rb[cb]]} += \text{spinReconstructDir1Minus(u[1]*tmp2)}; \)

7. \( \text{tmp[rb[otherCB]]} = \text{adj(u[0])*spinProjectDir0Plus(psi)}; \)
8. \( \text{tmp2[rb[cb]]} = \text{shift(tmp, BACKWARD, 0)}; \)
9. \( \text{chi[rb[cb]]} += \text{spinReconstructDir0Plus(tmp2)}; \)

Note that in the cases with \texttt{BACKWARD} shifts, the \( u \) values are taken from the \texttt{otherCB} checkerboard, while they otherwise come from the \texttt{cb} checkerboard. Thus in Listing 7.2, the \texttt{u[0]} on line 3 is a different \texttt{u[0]} than on line 17. The \texttt{adj} function returns the adjoint of a matrix.

There is one additional parameter to the Dsslash procedure called \texttt{isign}. When \texttt{isign=1}, the above sequence of operations is performed. When \texttt{isign=-1}, the “Plus” and “Minus” spin projection and recombination functions are interchanged.

### 7.1 Decomposition and Communication

At this stage \( L \) is redefined to specify only the lattice size in the spatial dimensions, and denote the size in the time dimension as \( T \), such that the lattice has \( L^3T \) sites. A
1D block decomposition in the time dimension is used to divide the lattice, and thus the work, across the SPEs. This is inherently load-balanced as long as $T$ is a multiple of the number of SPEs. This is not a problematic requirement, since existing lattice QCD simulations frequently use power-of-two sizes. A chunk allocated to an SPE is illustrated in Fig. 7.1. For convenience, let $T_{\text{chunk}} = T/N_{\text{spus}}$.

Having assigned a $L^3 \times T_{\text{chunk}}$ chunk to each SPE, the data required by each SPE for the Dslash operation is as follows, derived from the data dependencies in Listing 7.2. Sizes are given in spinors ($4 \times 3$ complex arrays) and colour matrices ($3 \times 3$ complex arrays).

- Input: Psi on sites in the chunk but on the opposite checkerboard to the sites for which Chi is calculated. $L^3 T_{\text{chunk}}/2$ spinors.
- Input: Psi on the 3 dimensional slice one lattice space forward in the time direction from the forward edge of the chunk. $L^3/2$ spinors.
- Input: Psi on the 3 dimensional slice one lattice space backward in the time direction from the backward edge of the chunk. $L^3/2$ spinors.
- Input: U[0..3] values on both checkerboards in the chunk. $4L^3 T_{\text{chunk}}$ colour matrices.
- Input: U[3] values on the 3 dimensional slice of lattice space backward in the time...
direction from the backward edge of the chunk, on the opposite checkerboard for the sites for which Chi is calculated. $L^3/2$ colour matrices.

- Output: Chi on the requisite checkerboard within the chunk. $L^3T_{chunk}/2$ spinors.

All of the above are stored sequentially in main memory when using the default QDP++ rb2 indexing scheme, since the slowest varying index is the checkerboard (0 or 1), and the next slowest varying is the time coordinate. The calculation of an element of Chi on the SPE requires the surrounding Psi values. To locate these values in the transferred buffers, a series of index manipulations are performed: first the Chi coordinate is determined in the global lattice space, then this is transformed into an index in the global lattice storage, and finally this is transformed into an index in the local storage. This does not need to be particularly efficient, since it is done once at initialisation, and pointers to the data required for each Chi element are stored. This sacrifices local store space for computational efficiency and the development time saved by not having to optimise the index manipulations. The requirement for local store could be reduced by a factor of two by storing unsigned short int (2 bytes) array indices rather than pointers (4 bytes). However, using 2-byte integers might generate extra shuffle instructions when the indices were dereferenced.

A beneficial characteristic of the Dslash procedure is that it is typically used repeatedly for the same value of the gauge field U, meaning that the U data need only be transferred once. As such, the time taken to transfer U is negligible when the procedure is run many times. As such, the data transferred at each Dslash call is limited to the Psi input and Chi output. Analysis of the communication patterns within the input arrays is complicated somewhat by the fact that not all of the buffers are accessed in the same sequence. As shown in Fig. 7.2, calculation of each element of Chi involves values from various locations in the Psi arrays (the main chunk, the forward slice and the backward slice). This plot is not intended to be straightforwardly interpreted, rather to illustrate the complexity associated with predicting accesses in checkerboard indexing schemes, which makes any kind of double buffering communications more difficult.

### 7.2 SIMD optimisation

SPE arithmetic operations act on 16B vectors, corresponding to four single precision floating point numbers. Operations on scalar values are slow, since not only is a full vector calculation done even if only one scalar result is needed, the scalar must be shifted to the correct position within a vector register prior to the operation. For this reason, all arithmetic should be performed using SIMD vector instructions where possible.

The SPE load/store instructions also operate on 16 byte vectors, which may be loaded and stored only from 16 byte aligned locations in the local store. To load a vector spanning a 16 byte boundary, two vector loads are required, followed by a shuffle operation.
to compose the desired vector in a register. This is vastly slower than loading a properly aligned vector, which is accomplished in one 6-cycle load instruction.

The peak performance of the SPE is realised when one pipelined vector multiply-add instruction is issued per cycle. The multiply-add instruction has the same cost as a single add or multiply, so when doing a multiply an extra add is effectively free, and vice-versa. Equally, if a calculation involves both multiplies and adds, they should be combined into a fused multiply-add operation where possible.

SIMD operations are exposed to the developer using compiler intrinsics which operate on so-called “vector” datatypes\[15\], generally mapping directly to machine instructions. For example, two vectors of floats may be multiplied together as follows:

```c
vector float a;
vector float b;
vector float c;
c = spu_add (a, b);
```

This is fairly straightforward to read, although clearly produces code which compiles and runs only on the SPE. The SDK includes a header named `vec_literal.h` which provides macro abstractions of the vector operations which resolve either to SPE intrin-

\[2\]The QDP++ REAL type is defined using `typedef`. Unfortunately `typedefs` are not resolved before the compiler evaluates a type name like `vector REAL`. To work around this, `#define REAL float` is always written before SIMD code which uses the REAL type.

Figure 7.2: Data dependencies in Psi arrays for the calculation of each index of Chi
sics or PPE Altivec intrinsics. However, this is not necessary here since all arithmetic is performed on the SPE.

In addition to the well known load, store and arithmetic operations, “shuffle” operations are used extensively in SIMD programming. The shuffle instruction creases a vector from bytes selected from two existing vectors. For example, if one had eight floats and wanted a vector consisting of the first and last two, then one might use the spu_shuffle intrinsic as shown in Listing 7.4.

In refactoring the Dslash calculation to SIMD form, three main stages are identified: the spin projection, multiplication by the color matrix, and spin reconstruction. Each spin projection and reconstruction function has 8 variants, corresponding to forwards and backwards in each dimension. Although mathematically each of these variants is simply a matrix multiplication by a different matrix, in practice they are programmatically distinct since the matrices in question are small and sparse so the implementation of the multiplication is unrolled into the instructions required for the non-zero elements. The color matrix multiplication has two variants, one for multiplying a vector by the color matrix and another for multiplying by its adjoint. Each of these 10 operations is hand-translated from scalar code in QDP++ to SPE intrinsics for the Cell Dslash.

As an example of the optimisation process, a “forward spin projection” in dimension 0 is considered, called SpinProjDir0Plus in QDP++. Listing 7.5 is the generic implementation found in QDP++, which is already not very readable. Compiling this using spuxlc -O3 -qhot and using cycle counters in the Cell simulator gives a timing of 141335 cycles for n_vec=1024. Rewriting the arithmetic as a series of spu_madd and spu_shuffle calls gives a modest runtime decrease to 98291 cycles. Using “register vector float” type temporaries to explicitly load from src reduces the timing further to 49497 cycles. Finally using temporaries to perform explicit stores of the result to dst reduces the duration to 21084 cycles, or about 21 cycles per spinor. In the code generated from the final version, all the multiply-add, shuffle, load and store instructions are pipelined. The final SPE code is shown in Listing 7.6. Note the liberal use of the register modifier on local variables: the large size of the SPE register file makes it possible to do this without worrying about running out of registers.
Listing 7.5: QDP++ generic scalar spin projection function

```c
inline void inlineSpinProjDir0Plus(const REAL* src, REAL *dst, unsigned int n_vec) {
    REAL tmp_spinor[4][3][2];
    const REAL* src_shadow = src;
    REAL* dst_shadow = dst;
    const int re = 0;
    const int im = 1;
    const int Ncmpx = 2;

    for(unsigned int site=0; site < n_vec; site++) {
        REAL* tmp_shadow = &(tmp_spinor[0][0][0]);

        // Stream in the spinor
        for(int stream=0; stream < Ns*Nc*Ncmpx; stream++) {
            *(tmp_shadow++) = *(src_shadow++);
        }

        // Project and store
        // Spin 0
        for(int col=0; col < Nc; col++) {
            *(dst_shadow++) = tmp_spinor[0][col][re] - tmp_spinor[3][col][im];
            *(dst_shadow++) = tmp_spinor[0][col][im] + tmp_spinor[3][col][re];
        }

        // Spin 1
        for(int col=0; col < Nc; col++) {
            *(dst_shadow++) = tmp_spinor[1][col][re] - tmp_spinor[2][col][im];
            *(dst_shadow++) = tmp_spinor[1][col][im] + tmp_spinor[2][col][re];
        }
    }
}
```
Listing 7.6: Cell SIMD spin projection function

```c
void cell_SpinProjDir0Plus(const REAL* src, REAL *dst, unsigned int ←
n_vec)
{
    const vector REAL *src_v = (vector REAL *) src;
    vector REAL *dst_v = (vector REAL *) dst;

    // Pick out the 4th, 3rd, 6th and 5th floats
    register vector unsigned char Perm = (vector unsigned char)
        {12, 13, 14, 15,
         8, 9, 10, 11,
         20, 21, 22, 23,
         16, 17, 18, 19};

    register vector float oddsevens = (vector float) {-1, 1, -1,←
        1};

    // Temporaries for shuffled inputs
    register vector float shuf_lat, shuf_2nd, shuf_3rd;

    // 6 registers representing a 24 element spinor
    register vector float tmp0, tmp1, tmp2, tmp3, tmp4, tmp5;
    // 3 registers representing a 12 element half-spinor
    register vector float dst0, dst1, dst2;

    for(int site=0; site < n_vec; site++) {
        tmp4 = src_v[4];
        tmp5 = src_v[5];
        tmp0 = src_v[0];

        shuf_1st = spu_shuffle (tmp4, tmp5, Perm);
        dst0 = spu_madd (oddsevens, shuf_1st, tmp0);

        tmp3 = src_v[3];
        tmp1 = src_v[1];

        shuf_2nd = spu_shuffle (tmp5, tmp3, Perm);
        dst1 = spu_madd (oddsevens, shuf_2nd, tmp1);

        tmp2 = src_v[2];

        shuf_3rd = spu_shuffle (tmp3, tmp4, Perm);
        dst2 = spu_madd (oddsevens, shuf_3rd, tmp2);

        dst_v[0] = dst0;
        dst_v[1] = dst1;
        dst_v[2] = dst2;

        src_v += 6;
        dst_v += 3;
    }
}
```

55
In place of the scalar code’s series of scalar adds and substracts, the SIMD version in Listing 7.6 uses the vector multiply-add instruction, with one factor being the original add/subtract operands, and the other being an array of +1 and -1 values. Recall that this additional multiplication is free, since a multiply-add costs the same as an add. These multiplications are not counted as part of the Flop/s performed by the functions, since they are not a necessary part of the calculation being performed and were not present in the original code. As such, the three multiply-add instructions are counted as 12 Flops.

Using this scheme, our vectorised spin projection function obtains a performance of 1864MFlop/s, compared to the original scalar code’s 278MFlop/s.

Comparing Listing 7.5 with Listing 7.6 makes it apparent that the process of optimising for the Cell is utterly destructive of the original, producing code whose operation is difficult to understand by inspection. As such, these Cell-optimised versions must always be maintained separately to generic code. This fits into the existing structure of QDP++, which already includes versions of key functions optimised for SSE instructions, in addition to the generic implementation.

The execution of the optimised spin projection function may be plotted using the asmVis tool[56], which provides a simulated execution of SPE code along with a visualisation to aid the optimisation process. A screen shot is shown in Fig. 7.3.

---

### Figure 7.3: asmVis displaying the SPE optimised spin projection function (the surrounding loop over sites is removed). The left-hand column of instructions shows the arithmetic pipeline, while that on the right shows the logic pipeline. “X” characters indicate work being done, while “|” characters indicate a pipeline stall.
The longer and more computationally intensive matrix-vector multiplication functions lend themselves even better to optimisation in the SPE. The SPE optimised implementation does two multiplications at once, since the input and output vectors are complex 3 vectors (24 bytes), and best performance is achieved on multiple-of-16-byte arrays. The optimised version performs two matrix-vector multiplications in about 83 cycles according to asmVis, which is equivalent to about 5GFlop/s (recall that this is only one of eight SPEs).

### 7.3 Further serial optimisation

The SPE register file consists of 128 registers of 16B each, all of which are accessible to the floating point unit. This unusually large quantity of register storage is easily enough to accommodate a color matrix (5 registers), a spinor (6 registers) and many temporaries. However, the compiler does not always take advantage of this, performing unnecessary loads and stores, through insufficient analysis or strict aliasing rules. Particularly, it does not do a good job of keeping values in registers between function calls when performing inlining. To obtain the best possible performance this is overcome by fusing functions which re-use the same data. Rather than having separate projection, matrix multiplication and reconstruction functions, the three are combined, such that all three operations can share the same register temporaries. This requires 8 fused functions, one for each direction. Since the color matrix multiplication functions (plain and adjoint) are each common to 4 of the fused functions, they are made into pre-processor macros, operating on register vectors floats defined in the scope into which they are included. Although this is substantially less elegant and flexible than having separate functions which the compiler inlines, in practice it does give a measurable increase in performance, doing a better job than the compiler at minimizing loads and stores.

### 7.4 Padding color matrix arrays

Since a color matrix (a $3 \times 3$ complex matrix) has size 72B in single precision, if the first element in an array of color matrices is aligned to a 16B boundary then the subsequent element would be offset by 8B, $(72 \% 16 = 8)$. This causes a problem for SIMD operations, as explained in section 7.2. To solve this, one may either check the alignment of a color matrix prior to operating on it, and copy it out to an aligned location if necessary, or pad each element by 8 bytes.

Since the color matrices are loaded only once at initialisation, the cost of padding the color matrix arrays on the SPE is negligible when Dslash is called many times. Conversely, the cost of making aligned copies when operating on the color matrix is incurred at each invocation of the Dslash procedure. The padding clearly incurs a penalty in the local store space required: an increase from 72 to 80 bytes per color matrix, or 11%.
The color matrices constitute the majority of the local store space used for data, so this increase is significant to the overall local store requirement. This is only a worthwhile sacrifice as long as the serial performance of operations on the SPE is a bottleneck: if the procedure were communications-limited then it would might be preferable to make per-operation aligned temporaries of an unpadded array and thus accommodate a larger sub-lattice.

7.5 Generalisation for isign and cb parameters

The work described in this chapter so far considers working on only one half of the rb2 checkerboard at a time, for example cb=0. Performing the calculation for cb=1 is identical, apart from taking the input and outputs from different locations in main memory (recall that in QDP++ default storage format the CB is the slowest varying index, such that the first half of a lattice-wide array in main memory is the 0 CB and the second half is the 1 CB). This dependency also applies to parts of the U data loaded on the SPEs, meaning that U communications would be required at each Dslash call for a different CB than the last.

To overcome this problem, one may exploit the way the Dslash operator is typically used: that is, assume that the user will always call Dslash for CB 1 immediately after calling it for CB 0. Given this, the Dslash function presented to the user can be defined to calculated both CB at the same time, which keeps management of the checkerboarding internal to the implementation. To calculate both checkerboards simultaneously, half of the SPEs are assigned to work on CB 0, and half to working on CB 1. Thus each SPE has constant U arrays, appropriate to his own chunk and CB. Clearly this implementation would only give half performance if the user required only one CB. To provide best performance without making the assumption of both CB being calculated at the same time, it would be necessary to conditionally copy out parts of U at each call. This might be accomplished without seriously degrading performance, if these communications were overlapped with computation.

Splitting the SPEs in two by CB affects the constraints on what lattice may be accommodated on a given number of SPEs: it is required that double the lattice size in the time dimension be a multiple of the number of SPEs. This means that a lattice of size 4 in the time dimension which previously would not run on 8 SPEs (since each SPE must have at least one time ordinate) now works.

The isign parameter is straightforwardly accommodated: it requires that where for isign=-1, “Plus” spin projection/reconstruction operations were performed, for isign=1 “Minus” operations are performed and vice versa (refer to Listing 7.2 which is for isign=0). This is complicated slightly by the choice to fuse the projection/recombination functions with the matrix multiplication macros, since with isign=1 the adjoint form of multiplication is associated with the “Plus” spin operations, whereas when isign=-1 the adjoint form is instead used with the “Minus” spin operations.
This may be worked around by creating alternative forms of the fused operations for use when \( \text{isign} = -1 \). This has the disadvantage of increasing the maintenance cost of the code, as well as increasing the size of the compiled code. The maintenance issue could be mitigated somewhat by refactoring the fused operations into preprocessor directives and invoking them twice with differently defined matrix multiplication functions. In this case the simpler approach of duplicating the functions was chosen, since this is intended as only proof-of-concept code. Of course, the issue could be completely eliminated by reverting the decision to use the fused functions, but that would be contrary to the goal of writing this implementation for maximum performance.

7.6 Performance

The performance of the Cell Dslash is evaluated using the test program \( t_{\text{dslashm\_cell}} \), included as appendix E.3. The results are quoted in GFlop/s, where the operation is taken to perform 1392 floating point operations. This count is the same used in the Dslash test program \( t_{\text{dslash}} \) included in QDP++, and is an underestimate of the actual number of floating point operations performed in the Cell version, since the vectorised SPE code performs many more multiplications than the original code, to set the sign of numbers.

Performing typical strong scaling benchmarks is complicated by the limitation on the data size which can be accommodated in the local stores. A system size which will run on 2 SPEs is somewhat inadequate to test the performance of 16. For this reason a variety of system sizes were used, some providing coverage of low numbers of SPEs, some testing larger numbers of SPEs. In addition, weak scaling was measured using a system size with \( L = 2 \) and \( T \) proportional to the number of SPEs used. This provides a direct test of the synchronisation and communications, since the amount of serial work per SPE remains constant.

The results of these benchmarks are shown in Fig. 7.4. Each series shows a different system size, including the weak scaling case in which the system size is proportional to the number of SPEs. In the following discussion, system sizes are expressed as \( (L, T) \) for brevity.

The weak scaling case is shown in more detail in Fig. 7.5, where it is compared with a straight line between the \( N_{\text{spus}} = 2 \) point and the origin. There is close to linear scaling up to 8 SPEs, which falls away noticeably for greater numbers of SPEs. This corresponds to the regime in which some of the SPEs are on a separate chip, with a connection between the two EIBs provided over a 25.6GB/s I/O port. The almost-linear weak scaling plot indicates that the performance is limited largely by serial performance.

The \( (6, 4) \) system is shown in Fig. 7.4 as a single datapoint, as it runs only on exactly 8 SPEs: a smaller number of SPEs would not provide sufficient LS capacity, and to use more would require a larger size in the time dimension in order to provide each SPE with at least one 3D slice. This system is of particular interest in the context of
a multi-node system, along with the \((4, 24)\) system from the weak scaling series. The \((4, 24)\) system gives higher performance, but it is quite limited in this whole-lattice time dimensions possible: only systems with multiple-of-24 time lengths would be possible. The \((6, 4)\) system uses only around half of the available LS, but would be much more flexible in terms of simulations on different \(T\) sizes across multiple nodes. A good compromise might be possible if some quantity of the colour matrix arrays were streamed from main memory at each iteration, slightly decreasing the local store space requirement and allowing a \((6, 8)\) system on one Cell processor, at the cost of increased main-memory I/O.

Smaller systems which can be run on smaller numbers of SPEs are also shown on Fig. 7.4. These scale to smaller numbers of SPEs before performance starts to degrade. The only system included which can be run on the whole range of SPEs is the \((2, 48)\) size, which is small enough to run on 2 SPEs, but sufficiently long in the time dimension to run on 16 SPEs. Performance of this system falls off from 6 SPEs, although still reaches 30GFlop/s on one Cell processor (8 SPEs), which remains impressive in comparison with conventional processors. The \((4, 8)\) system achieves close to 40GFlop/s on 8 SPEs, making it a reasonably efficient choice of system size which could be useful in multi-node system for constructing power-of-two-sized global lattices.

Table 7.1 shows a comparison of the Cell to other platforms performing the Dslash operation in single precision. The \((6, 4)\) lattice allows the conventional processors to work in L2 cache, since it has a total size of 145kB (1 spinor per site, 4 colour matrices per site). A larger lattice would cause the conventional platforms to work out of main memory, while the Cell version would simply not be capable of it. However, if the Cell
Figure 7.5: Comparison of Cell Dslash weak scaling with ideal linear scaling

<table>
<thead>
<tr>
<th>Processor</th>
<th>Compiler</th>
<th>Implementation</th>
<th>GFlop/s ((6, 4) lattice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell 3.2GHz</td>
<td>XLC -O4 -qhot=simd</td>
<td>Cell Dslash</td>
<td>41</td>
</tr>
<tr>
<td>Power5+ 1.5GHz</td>
<td>XLC -O4 -qhot</td>
<td>Generic QDP++ Dslash</td>
<td>0.11</td>
</tr>
<tr>
<td>Xeon 2.4GHz</td>
<td>gcc -O3</td>
<td>intel_sse_wilson_dslash</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Table 7.1: Comparison of Cell Dslash with other architectures

version were adapted to stream parts of the dataset out of main memory then it would probably remain competitive with conventional platforms due to the Cell’s relatively high memory bandwidth.

The Power5+ benchmark was performed on the HPCx system, using the generic QDP++ reference Dslash. The sustained performance of 0.11GFlop/s is poor in comparison to the other processors, demonstrating that the XLC compiler used does not adequately optimise the naive Dslash code on Power5+.

As an example x86 system, a 2.4GHz Xeon system is used, along with a high-performance SSE implementation of the Dslash operation encapsulated in a library called intel_sse_wilson_dslash. This is bundled into the Chroma tarball and optionally compiled in on systems which support SSE. This provides an impressive performance of 4.5GFlop/s, demonstrating the high performance possible with this algorithm using low-level SIMD programming. However, the performance of the Cell implementation presented here is almost 10 times greater than even the SSE-optimised version on the

3Streaming SIMD Extensions (SSE) are vector extensions to the x86 architecture present on recent Intel
Xeon.

The performance of the Cell implementation is such that a Dslash over a $(6, 4)$ sub-lattice takes $33\, ns$ per site, or $28.5\mu s$ in total. Using existing benchmarks of off-the-shelf interconnects[58], one can compare this on-chip runtime to inter-node communications in a multi-node system. The best off-the-shelf interconnects offer latencies of the order a few $\mu s$, while the size of the halo data for a sub-lattice this size is of the order tens of kB, which may be expected to incur a communications time around $10\text{-}20\mu s$ in the best case. This communications time is of the order of the serial communications time, so for good sustained performance in a multi-node system it would be imperative to overlap off-node computation and calculation.

7.7 Future work

Due to the time constraints on this project, many avenues remain unexplored in the broader question of exploiting the computational power of the Cell for LQCD, as well as practical issues which would need to be addressed before putting a Cell Dslash into production use.

- For simplicity of the PPE-side interface, the implementation presented here simply initialises SPE-side U arrays at the first call to Dslash, and they remain constant for the duration of execution. Real codes would need multiple Dslash operations with different U (while still using each U many times), thus this part would have to be moved out into an explicit start-up function. This is how the existing SSE version (intel_sse_wilson dslash) operates. Such a change would be trivial to implement.

- Transfers larger than 16kB are currently performed by enqueuing multiple DMA transfers of up to 16kB each. Alternatively, one could compose a DMA list of such transfers and reduce overheads by generating this list at startup and enqueuing it at each iteration of the algorithm. This may or may not yield superior performance.

- All work here was done in single precision, for reasons explained in chapter 4. Short of maintaining separate DP and SP versions of the code, it may be possible to write more general arithmetic code if the compiler optimisation is well understood, such that a single version could switch between SP and DP. Even if results obtained in SP were determined to be satisfactory to users of the code, DP functionality would at least be required for comparison to make such a determination on a Cell machine.

- Currently there is a fairly low ceiling on the size of sub-lattice which may be accommodated on each SPE. This limit could be made more flexible at the cost of increased communications by streaming in part or all of the U arrays at each iteration rather than requiring the whole dataset to fit into the LS in one go. Equally,
the Psi and Chi data could be treated in a streaming manner rather than having full-sized local arrays in the LS for the whole sub-lattice assigned to the SPE. Some or all of the added communications cost could be hidden using double-buffering. This increased flexibility would be particularly relevant in a parallel machine which may not necessarily have enough processors for each Cell to have such as small portion of the lattice as is currently required.

- If a system composed of dual-processor Cell blades is under consideration, then it would be useful to know whether greater performance is obtained from running a process using 8 SPEs on each Cell, or running a single process using all 16 SPEs. The former cannot be reliably predicted by multiplying single-cell performance by two, since 8 SPE runs take advantage of the operating system running on the idle processor. There would also be system-specific memory issues since each processor has its own memory controller.

- In the current code, communication in the T dimension is implicit in the distribution and collection of the fermion (Chi and Psi) data from and to main memory at each Dslash call. However, if no further manipulations to the data are required in between Dslash calls then it would be more efficient to keep this data on the SPEs. In this case, it would be necessary to implement halo swapping in the T direction in between the SPEs. Once this were the case, the PPE would be essentially “out of the loop” apart from setting the SPEs running for a given number of iterations. In this case, it may be possible to save some amount of synchronisation runtime by implementing a peer-to-peer synchronisation between SPEs while iterating the Dslash procedure. A higher performance synchronisation may also be motivated by more frequent use, if off-node communications demanded multiple synchronisation points during each Dslash procedure.

- This implementation is purely scalar in nature, in the sense that the level of parallelisation is limited to a single cell blade. If the Cell were to be used “in anger” for LQCD then the code would have to interleave off-node communications with on-Cell calculations and communications. The current on-Cell parallelisation whereby the division between SPEs is in the T dimension could be retained in a multi-node version of the code, keeping the T dimension on-node and diminishing requirements for the inter-node interconnect to a 3D torus. If the dataset was permanently distributed across SPEs rather than being scattered and gathered at each iteration, it would be natural to have the off-node communications controlled by the SPE threads, rather than involving the PPE. In fact, given that the the Cell has I/O interfaces directly connected to the EIB, it is possible that the network interface would be directly connected to one of these, meaning that the data-flow could pass directly from the SPE to the network card, bypassing main memory entirely, and ending up directly on an SPE on another node. Whether this could be implemented effectively is a question for those engineering the communications hardware and software stack.
Chapter 8

Discussion

Chapters 4 and 5 provided valuable insight into the methods available to program the Cell processor. However, the approach used – encapsulating Cell-specific code within a single operation such as vaxpby – turned out to place a rather low cap on performance (up to 6GFlop/s), due memory bandwidth constraints. Heavier modification of QDP++ to overcome the memory bandwidth problem by persistently distributing lattice data across SPEs would be a substantially greater software engineering challenge, much more so than encapsulating and Cell-optimising only the most computationally expensive part of a LQCD calculation, as was done in chapter 7.

The attempt to create a high performance Dslash for the Cell was a great success, with performance up to 45GFlop/s obtained on a single Cell processor. This was accomplished only in single precision, on a processor with a 200GFlop/s peak in single precision. Future revisions of the Cell processor featuring 100GFlop/s double precision peak may be expected to run a double precision adaption of the Dslash presented here at around 20GFlop/s, which remains a substantial improvement on conventional processors.

The Dslash implementation presented here represented a highly performance-centric approach, paying no regard to readability and maintainability of the code. It represents a branch of parts of the QDP++ code, analogous to the existing SSE Dslash library. In particular, the SIMD optimisation performed on the SPE code was destructive of the original code structure, and resulted in largely unreadable code. The various approaches explored in the first part of this report provided more of a compromise between readability and performance. In particular, RapidMind achieved impressive performance, reaching the memory bandwidth limit in spite of being almost trivial to use. However, work on the Dslash kernel required great attention to serial performance on the SPEs as well as parallelisation efficiency, and RapidMind does not provide the opportunity to write such low-level optimised SPE code as was needed in the Dslash kernel.

ALF was found to be disappointing, offering neither much abstraction from the details of the parallelisation, nor performance as high as the hand-coded libspe2 version. Although it was possible write a functional ALF program using only a simple kernel
function on the SPE side, best performance was found to require hand-coded data transfer functions on the SPE, demanding that the programmer be quite familiar with Cell effective address manipulation and the internal structure of the SPE-side ALF buffers. Future versions of ALF may resolve some of these issues. If multi-use workblocks were used, then ALF could be expected to perform similarly to the libspe2 code, at the cost of providing essentially no higher-level abstraction compared to hand-coded parallelism using libspe2.

Similarly, libcache was found to offer the kind of degraded performance that one would expect from calling a library function at every load calculation. The libcache computational kernel still requires libspe2-based thread management and synchronisation to go with it, rendering it barely easier to program with than pure libspe2 and SPE intrinsics. However, libcache could still be very useful for algorithms requiring random access to large datasets, as opposed to the predictable access patterns in the test cases used in this project. Once again, future revisions of the Cell SDK may provide improvements to the performance issues encountered here.

To some extent, the experiences of this project apply to using the Cell for scientific programming in general. Very many problems are solved using linear algebra as LQCD is, and similar issues would be expected with all such codes. Codes not based on linear algebra operations may be more difficult to cast into the SIMD form required for high serial performance on the SPE, causing a greater development time for the SPE kernels, or in some cases preventing good SPE performance entirely. Similarly, some problems with unpredictable data access patterns may lend themselves more to a cache-based processor than the the Cell processor, even when using libcache on the Cell (since a hardware cache imposes lesser overheads).

As a test-case, QDP++ is somewhat unusual amongst open scientific codes in that it is written in a object oriented fashion, making heavy use of C++ templates. Many codes are not so well engineered, and may not make the encapsulation of operations so readily possible. However, whilst the QDP++ design presents a very neat interface to the application programmer, it does complicate the code in the library, and the process of locating the implementation of a given operation was made more difficult by the need to work through several layers of templates. When the ultimate approach is to be a low-level one as it currently is when programming the Cell for high performance, a more old-fashioned code operating directly on arrays might be more accessible to a developer porting to Cell, despite being somewhat less favourable from a software engineering standpoint.

Although this report has considered only issues associated with running LQCD calculations on one or two Cell processors, ultimately applying the Cell processor to LQCD or similar problems would demand a communication system that could keep up with the performance obtained on a single cell blade. The main Cell-specific issue in this situation is that for best performance any interconnect should be directly accessible from the SPEs, rather than requiring the involvement of the PPE, or worse the use of buffers in main memory. This is in principle possible, since the Cell processor provides ample I/O bandwidth on the EIB, which in the current Cell blade platform is connected indirectly.
to an InfiniBand interface via PCI Express. To determine whether it would be possible to implement direct off-node communications from SPE LS using this kind of system would be an interesting avenue of future investigation. Alternatively, a communications device could be directly connected to the I/O interface which is currently used to connect to the other Cell processor, in a single-processor node.

A Cell-like ASIC adapted for LQCD might include an on-chip network interface as seen in QCDOC and recent advanced multi-core processors such as the Sun UltraSPARC T2[59]. However, such a chip does not seem feasible in the near-term: the custom QCDOC chip was based on general purpose components from IBM’s ASIC library with a few custom designed components, while the Cell is substantially unconventional design and is a tightly integrated custom system itself. As such, a modified Cell processor could be an expensive route compared to using existing Cell blade products or conventional MPP solutions[55]. Even current Cell-based products are not especially low-cost: QS20 Cell blades cost around $12,000[60], excluding interconnect. A cheaper Cell solution composed of gigabit Ethernet-connected PlayStation 3s is available for $18,000 for 10 nodes, but this would be useful only for loosely coupled applications[17].
Chapter 9

Conclusions

The most important outcome of this project was the performance result from chapter 7: a 45GFlop/s Wilson Dslash for the Cell processor. This work provided a proof-of-concept for running LQCD calculations on the Cell processor. Extensions to this could include a double precision version, which could be expected to achieve 20GFlop/s on the forthcoming double-precision Cell processor, and a multi-node parallel version of this code.

The results from chapter 5 justified the low-level approach taken with the Dslash kernel: Cell parallelism encapsulated at the level of single linear algebra operations was severely limited by memory bandwidth. With the exception of the RapidMind toolkit, the techniques evaluated failed to provide a significant abstraction from the hardware. For non-capability calculations on the Cell processor the RapidMind toolkit provided extremely straightforward and relatively efficient parallelisation across SPEs. Nevertheless, to obtain best performance a gruelling hand-optimisation of the SPE code is necessary, which is found to render substantially better performance than the optimising compiler, although the Cell compiler is relatively immature and there is certainly scope for improved code generation in future revisions.

The Cell processor remains challenging to the scientific application programmer, demanding significant platform-specific knowledge to obtain good performance. However, for applications concentrating work in a single kernel – as LQCD codes do in Dslash – the effort involved in producing a Cell-optimised version of the kernel can render an order-of-magnitude performance improvement over conventional platforms.
Bibliography


Note: some of the above user guides and handbooks are part of the Cell SDK version 2.1, available online at time of writing at http://ibm.com/developerworks/power/cell/downloads.html
Appendix A

Definitions

The following definitions are used throughout this report:

*The SDK*  The official Cell SDK version 2.1 downloaded from ibm.com

*QDP++*  The qdp++-1.23.2 release tarball downloaded from jlab.org

*Cell blade*  The IBM QS20 REV3 blade.

*Flop/s*  Floating point operations per second, where such metrics refer to the logical operations performed in an algorithm, rather than the literal number of floating point operations performed by the processor. This distinction is necessary to ensure a fair comparison between scalar and vector codes.

*SIMD*  Single Instruction Multiple Data

*API*  Application Programmer Interface

*ASIC*  Application Specific Integrated Circuit

*SPE*  Synergistic Processing Element

*PPE*  Power Processing Element

*B*  Bytes

*b*  bits
Appendix B

Building

B.1  Cross-compiling libxml2 for Cell

```bash
# Compiling libXML2 for PPE on a FC6 x86 machine:

$ export CC=/opt/cell/bin/ppu32-gcc
$ export LDFLAGS="-L/opt/ibm/cell-sdk/prototype/sysroot/usr/lib"
$ ./configure --host=powerpc --prefix=/home/jspray/QDP-Cell/local --without-python

# Edit libtool to replace all instance of ranlib with /opt/cell/bin/ppu-ranlib

cat libtool | sed 's/ranlib/\|opt/cell/bin/ppu-ranlib/' > libtool.fixed
mv libtool.fixed libtool

$ make
$ make install
```

B.2  Cross-compiling QDP++ for Cell

```bash
# Compiling QDP++ 1.23.2 for Cell on a FC6 x86 machine
# Using ccache for faster recompilation

$ export CC="ccache/opt/cell/bin/ppu-gcc"
$ export CXX="ccache/opt/cell/bin/ppu-g++"
$ export CXXFLAGS="-pthread -O3"
$ export LDFLAGS="-L/opt/ibm/cell-sdk/prototype/sysroot/usr/lib64 -L/home/jspray/prototype-2.1/sysroot/usr/lib64 -R/opt/ibm/cell-sdk/prototype/sysroot/usr/lib"
$ export CELL_TOP="../"
```
$ ./configure --host=powerpc --enable-parallel-arch=scalar
   --with-libxml2=/home/jspray/QDP-Cell/local --prefix=/
   home/jspray/QDP-Cell/local

$ for file in 'find ./ -name Makefile' ; do cat $(file) |
   sed 's/ranlib/\opt/\cell/\bin/\ppu-ranlib/' > $(file)
   .new ; mv $(file).new $(file) ; done

$ make
Appendix C

Modifications and additions to QDP++

This section is included to provide a brief overview of how the modifications made here fit into the QDP++ library. The code, consisting of benchmarks and proof-of-concept code is somewhat “rough around the edges”. Cell code is found in three directories of the qdp++ source tree: examples, include, lib, and spu. Those files having Cell-specific content are listed in the following tree:
The lib/cell-*.cc files contain most the PPE-side logic. The spu directory contains SPE programs: the vaxpby implementations are named blasops, blasops_alf and blasops_libcache. The Dslash SPE program is in spu/dslash.

Which vaxpby implementation to use is chosen in include/cell-config.h. The accompanying SPE programs should be uncommented in spu/Makefile, and make run in the spu directory before being running make t_blas_cell or so in the examples directory.

The SIMD code in the dslash SPE program is largely unreadable. To understand what operations are performed, the spin projection functions correspond to those in include/scalarsite_generic/generic_spin_proj_inlines.h, while the matrix-vector operations correspond to those in include/scalarsite_generic/generic_mat_vec.h.

Note that there are some extraneous elements of the code which do not directly relate to those results discussed in the report. Specifically, there is a spin_proj function in the blasops in spin_proj_alf SPE programs, and associated PPE-side logic for loading up the appropriate ALF task. The associated ALF task-switching is tested using examples/t_thrash.cc.
The machine gx260aj.epcc.ed.ac.uk, a Dell PC running Fedora Core 6, was used for compilation in this project, with SDK 2.1 installed in /opt. This was the only environment in which the code is known to compile correctly.
Appendix D

Variation of vaxpby performance with transfer size in libspe2 implementation
Figure D.1: Effect of DMA transfer size on performance of vaxpby3 operation for an $4^4$ lattice

Figure D.2: Effect of DMA transfer size on performance of vaxpby3 operation for an $8^4$ lattice
Figure D.3: Effect of DMA transfer size on performance of vaxpby3 operation for an $32^4$ lattice
Appendix E

Test Programs

E.1 residue.py validation script

This is a simple python script that calculates the root mean square (RMS) difference between two files containing lists of plain text numbers and reports whether it is greater or less than a give threshold.

Listing E.1: residue.py

```python
#!/usr/bin/python

import sys;
import math;

threshold = 0.00001

if (len(sys.argv) < 3):
    print "error: You meant to specify two files, didn’t you?"
    sys.exit(-1);

a = open (sys.argv[1]);
b = open (sys.argv[2]);
alines = a.readlines();
blines = b.readlines();
a.close ();
b.close ();

if (len(alines) != len(blines)):
    print "error: Line counts differ."
    sys.exit(-1);

numlines = len(alines);
afloat = 0.0;
bfloat = 0.0;
```

81
rms = 0.0;

for i in range(0, len(alines)):
    aitems = alines[i].split();
    bitems = blines[i].split();

    if (len(aitems) != 1 or len(bitems) != 1):
        print "error: Too many items on line", i
        sys.exit(-1);

    afloat = eval(aitems[0]);
    bfloat = eval(bitems[0]);
    diff = afloat - bfloat;
    rms = rms + diff * diff;

rms = math.sqrt(rms / len(alines));

print "RMS = ", rms

if (rms > threshold):
    print "threshold = ", threshold
    print \\x1b[31m\"Test FAILED\"\x1b[0m"
    sys.exit(1);
else:
    print \\x1b[32m\"Test passed\"\x1b[0m"
    sys.exit(0);

# Don’t write anything here, it won’t get executed

E.2 t_blas_cell

This program is a derivative of the t_blas.cc example included in QDP++.

Listing E.2: t_blas_cell.cc

// $Id: t_blas_cell.cc,v 1.2 2007/06/06 12:09:44 jspray Exp $
#include <iostream>
#include <iomanip>
#include <cstdio>
#include <time.h>
#include "qdp.h"
#include <blas1.h>

using namespace std;
using namespace QDP;
static int spacesize;
static int timesize;

void print_latticefermion (char *basename, OLattice<TVec> &z) {
    REAL *zptr = (REAL *) &z.elem(all.start()).elem(0).elem(0).real();
    int n_reals = (all.end() - all.start() + 1) * Ns * 6;
    char *dumpfile = (char*) malloc (sizeof(char) * 64);
    snprintf (dumpfile, 64, "%s_%d_%d_%d_%d.dat", basename, spacesize, spacesize, spacesize, timesize);
    printf ("print_latticefermion: writing %s\n", dumpfile);
    FILE *dump = fopen (dumpfile, "w");
    free (dumpfile);
    for (int i = 0; i < n_reals; i += (n_reals / 3)) {
        for (int i = 0; i < n_reals; i++) {
            fprintf (dump, "%.8f\n", zptr[i]);
        }
        fclose (dump);
    }
}

int main(int argc, char *argv[]) {
    // Put the machine into a known state
    QDP_initialize(&argc, &argv);
    
    /*
     * Command line options
     * "./t_blas_cell 10.0 4 2" gets you a 10 sec run on a
     * 4^3*2 lattice
     * 
     * Set QDP_CELL_TEST_LOGS to get the output written to files
     */
    //int icnt = 1;
    //if (argc > 1)
    //    icnt = atoi(argv[1]);
    //else
    //    icnt = 1; +/
    double timing_duration;
    if (argc > 1)
        timing_duration = atof(argv[1]);
    else
        timing_duration = 1.0;
if (argc > 2)
    spacesize = atoi (argv[2]);
else
    spacesize = 4;

if (argc > 3)
    timesize = atoi(argv[3]);
else
    timesize = 4;

printf (
    "%t_blas_cell:1+?iterations on %d*d%d*d%d lattice\n",
    spacesize,
    spacesize,
    spacesize,
    timesize
);

    int const dims[4] =
    {spacesize,
     spacesize,
     spacesize,
     timesize};

    // Setup the layout
    multi1d<int> nrow(Nd);
    nrow = dims;  // Use only Nd elements
    Layout::setLattSize(nrow);
    Layout::create();

    Real a = 1.5;
    Real b = -3.2;
    LatticeFermion qx;
    LatticeFermion qy;
    LatticeFermion qz;
    Double dnorm;
    Double dnorm2;
    gaussian (qx);
    gaussian (qy);

double tt;
QDPIO::cout << "calling V=aV+bV" << 1 << " times" << endl;
tt = QDP_AXPBY(qz, a, b, qx, qy, 1);
{
    double rescale = 1000*1000 / double(Layout::sitesOnNode()) / (double)1;
    tt *= rescale;
    int Nflops = 3*2*Ns*Nc;
    QDPIO::cout << "First iteration time (V=aV+bV)=" << tt << " micro-secs/site/iteration"
               << " Nflops = " << Nflops / tt << " Mflops" << endl;
if (icnt) {
    tt = 0;
    while (tt < timing_duration) {
        QDPIO::cout << "calling V=aV+bV" << icnt << " times" << endl;
        tt = QDP_AXPBY(qz, a, b, qx, qy, icnt);
        QDPIO::cout << "Ran " << icnt << " iterations in " << tt << " s/iteration\n";
        if (tt < timing_duration) {
            icnt *= (timing_duration / tt) * 1.1;
        }
    }
    double rescale = 1000*1000 / double(Layout::sitesOnNode());
    tt *= rescale;
    int Nflops = 3*2*Ns*Nc;
    QDPIO::cout << "Benchmark time(V=aV+bV)=" << tt
      << " micro-secs/site/iteration" << endl;
    if (getenv("QDP_CELL_TEST_LOGS"))
        print_latticefermion("vaxpby3", qz);
    QDP_finalize();
    exit(0);
}

E.3 t_dslashm_cell

This program is a derivative of the t_dslashm.cc example included in QDP++.

Listing E.3: t_dslashm_cell.cc

#include <iostream>
#include <cstdio>
#include "qdp.h"
#include "examples.h"
#include <sys/time.h>
using namespace QDP;

static int spacesize;
static int timesize;

void print_latticefermion (char *basename, LatticeFermion &z)
{
    REAL *zptr = (REAL *) &z.elem(all.start()).elem(0).elem(0)->.real();
    int n_reals = (all.end() - all.start() + 1) * Ns * 6;
    char *dumpfile = (char*) malloc(sizeof(char) * 64);
    snprintf(dumpfile, 64, "%s_%d_%d_%d_%d.dat", basename, spacesize, spacesize, spacesize, timesize);

    printf("print_latticefermion: writing\%s\n", dumpfile);
    FILE *dump = fopen(dumpfile, "w");
    free(dumpfile);

    Subset s = all;
    const int* tab = s.siteTable().slice();

    for(int j=0; j < s.numSiteTable(); j++) {
        int idx = tab[j];
        REAL *site_ptr = (REAL *)&(z.elem(idx).elem(0).elem(0).real());
        multi1d<int> coords = Layout::siteCoords(0, idx);
        fprintf(dump, "%.8f\%d\%d\%d\%d\%d\%d\(site\%d)\n", site_ptr[0], coords[0], coords[1], coords[2], coords[3], idx);
    }

    fclose(dump);
}

void print_latticefermion_comparison (char *basename, LatticeFermion &y, LatticeFermion &z)
{
    REAL *zptr = (REAL *) &z.elem(all.start()).elem(0).elem(0)->.real();
    int n_reals = (all.end() - all.start() + 1) * Ns * 6;
    char *dumpfile = (char*) malloc(sizeof(char) * 64);
    snprintf(dumpfile, 64, "%s_%d_%d_%d_%d.dat", basename, spacesize, spacesize, spacesize, timesize);

    printf("print_latticefermion: writing\%s\n", dumpfile);
    FILE *dump = fopen(dumpfile, "w");
free (dumpfile);

  Subset s = all;
  const int* tab = s.siteTable().slice();

  for (int j=0; j < s.numSiteTable(); j++) {
    int idx = tab[j];
    REAL *z_ptr = (REAL *)&(z.elem(idx).elem(0).elem(0).real());
    REAL *y_ptr = (REAL *)&(y.elem(idx).elem(0).elem(0).real());
    mult1d<int> coords = Layout::siteCoords (0, idx);

    fprintf (dump, "%.8f %.8f %d %d %d %d (idx ← %d),
     y_ptr[0], z_ptr[0], coords[0], coords[1], coords[2], coords[3], idx);

    if (fabs(z_ptr[0] - y_ptr[0]) < 0.0001)
      fprintf (dump, "MATCH\n");
    else
      fprintf (dump, "BREAK\n");
  }
  fclose (dump);

}

void print_coord_transform (char *basename, LatticeFermion &y) {
  int n_reals = (all.end () - all.start () + 1) * Ns * 6;

  char *dumpfile = (char*) malloc (sizeof(char) * 64);
  snprintf (dumpfile, 64, "%s_%d_%d_%d_%d.dat", basename, spacesize, spacesize, spacesize, timesize);

  printf ("print_coord_transform: writing %s\n", dumpfile);
  FILE *dump = fopen (dumpfile, "w");
  free (dumpfile);

  Subset s = all;
  const int* tab = s.siteTable().slice();

  for (int j=0; j < s.numSiteTable(); j++) {
    int idx = tab[j];
    REAL *y_ptr = (REAL *)&(y.elem(idx).elem(0).elem(0).real());
    mult1d<int> coords = Layout::siteCoords (0, idx);

    fprintf (dump, "%d %d %d %d (new_idx ← %d),
     coords[0], coords[1], coords[2], coords[3], new_idx);

    coords[0] += 1;
    if (coords[0] > spacesize - 1)
      coords[0] = 0;

    int new_idx = Layout::linearSiteIndex (coords);
    fprintf (dump, "%d %d %d (%d)\n", coords[0], coords[1], coords[2], coords[3], new_idx);
```c
int main(int argc, char **argv) {
    // Put the machine into a known state
    QDP_initialize(&argc, &argv);

    double timing_duration;
    if (argc > 1)
        timing_duration = atof(argv[1]);
    else
        timing_duration = 1.0;

    if (argc > 2)
        spacesize = atoi(argv[2]);
    else
        spacesize = 4;

    if (argc > 3)
        timesize = atoi(argv[3]);
    else
        timesize = 4;

    printf("t_dslashm_cell:\n",
           spacesize,
           spacesize,
           spacesize,
           timesize);

    int const dims[4] =
        {spacesize,
         spacesize,
         spacesize,
         timesize};

    // Setup the layout
    mult1d<int> nrow(Nd);
    nrow = dims;
    Layout::setLattSize(nrow);
    Layout::create();

    mult1d<LatticeColorMatrix> u(Nd);
    for(int m=0; m < u.size(); ++m)
        gaussian(u[m]);

    LatticeFermion psi, chi, tmp;
}```
random(psi);
LatticeFermion chi2;
LatticeFermion chi3;
random(psi);
chi = zero;
chi2 = zero;
chi3 = zero;
    int isign;
    int cb = 0;
{
    isign = 1;
    printf("Testing both cb\(dslash2)\n");
    dslash2(tmp, u, psi, isign, 0);
    chi2 = tmp;
    dslash2(chi2, u, psi, isign, 1);
    printf("Testing both cb\(dslash3)\n");
    // The zero is ignored.
    dslash3(chi3, u, psi, isign, 0);
    print_latticefermion_comparison("2vs3", chi2, chi3);
    }
    isign = 1;
    if (timing_duration > 0) {
        double tt;
        long long icnt;

        // Benchmark dslash3
        icnt = 1;
        tt = 0;
        while (tt < timing_duration) {
            printf("dslash3\%lld\ times\n", icnt);
            StopWatch swatch;
            swatch.start();
            for (int i = 0; i < icnt; ++i) {
                dslash3(chi3, u, psi, isign, cb);
            }
            swatch.stop();
            tt = swatch.getTimeInSeconds();
            printf("Ran\%lld iterations in\%fs,\%fs/\sim\iteration\n", 
                icnt, tt, tt / (double)icnt);
            if (tt < timing_duration) {
                if (tt > 0)
                    icnt *= (timing_duration / tt) * 89;
else
    icnt *= 2;
}

tt = 1.0e6*tt/((double)(icnt*(Layout::vol()/2)));
QDPIO::cout << "cb=" << cb << "isign=" << isign << endl;
QDPIO::cout << "The time per lattice point is" << tt << "micro sec" << "(" << (double)(2784.0f/tt) << "Mflops)" << endl;

// Benchmark dslash2
icnt = 1;
tt = 0;
while (tt < timing_duration) {
    printf("dslash2\%lld\times\n", icnt);
    StopWatch swatch;
    swatch.start();
    for (int i = 0; i < icnt; ++i) {
        dslash2(chi3, u, psi, isign, cb);
    }
    swatch.stop();
    tt = swatch.getTimeInSeconds();
    printf("Ran\%lld iterations in\%fs\%fs/iteration\n", icnt, tt, tt / (double)icnt);
    if (tt < timing_duration) {
        if (tt > 0)
            icnt *= (timing_duration / tt) * 1.1;
        else
            icnt *= 2;
    }
}

tt = 1.0e6*tt/((double)(icnt*(Layout::vol()/2)));
QDPIO::cout << "cb=" << cb << "isign=" << isign << endl;
QDPIO::cout << "The time per lattice point is" << tt << "micro sec" << "(" << (double)(1392.0f/tt) << "Mflops)" << endl;

// Time to bolt
QDP_finalize();
exit(0);