UNIVERSITY OF CALIFORNIA, SAN DIEGO

# CAFE: A Framework for Cell Application Development

A thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in

Computer Science

by

Joseph Keith Hammer, Jr.

Committee in charge:

Professor Scott Baden, Chair
Professor Sheldon Brown
Professor Geoffrey M. Voelker

2007

The Thesis of Joseph Keith Hammer, Jr. is approved:

_____

_____

_____
Chair

University of California, San Diego

2007

**Dedication**

*To my parents:*

*For my father, who always pushed me to be the best I could be...*

*For my mother, who always supported me in what I wanted to do...*

**Table of Contents**

**List of Figures**

**Acknowledgments**

First and foremost, I would like to acknowledge Kristen Kho for her valiant efforts in helping me write the CAFE framework and keeping me sane. Without her help I would most certainly still be coding or running around in circles trying to make another design decision. Thank you for being an invaluable pair programming partner and willing to jump in and hit the ground running with this project.

I would also like to acknowledge the team we worked with at IBM: Bruce D'Amora, Gordon Ellison, Sidney Manning, and Bob Szabo, for their assistance with our work and questions about the Cell BladeServer. I would also like to extend thanks to Rob Todd and Jessica Schupp for their additional support during the Cell Workshop at High Moon. IBM was the industrial sponsor for the UC Discovery Grant: "Implementing Cell Processor Engines for Resolving Constraints in Real-Time Graphic Environments" which funded this work.

I would like to thank CRCA Technical Director Todd Margolis for all of his efforts to setup and maintain the Cell BladeServer at UC San Diego. Also, I would like to extend thanks to Helena Bristow for all of her time and support.

Thank you to all of the members of the Experimental Game Lab team: Alex Dragulescu, Mike Caloud, Erik Hill, Carl Burton, and Daniel Tracy, for the great times over the last three years on the Scalable City project.

Also, I would like to thank the Research and Development Team from High Moon Studios: Noel Llopis, Jim Tilander, Charles Nicholson, and Rory Driscoll, for their patience in teaching me the ins and outs of real-world systems coding and

software engineering. Also, their advice on aspects of particular problems we ran into during development proved to be extremely valuable.

Finally, I would like to thank my advisors. Thank you to Professor Scott Baden, who taught me the fundamentals of parallel programming—without his abundance of patience and wisdom this could not have been possible. Thank you to Professor Sheldon Brown for believing in me over the last three years and allowing me to pioneer the development on the Cell for the gamelab. Thank you to Professor Geoff Voelker for the continuous flow of encouragement that started on the day I first met with him to ask about taking the big step into the Computer Science graduate program and never stopped.

ABSTRACT OF THE THESIS

CAFE: A Framework for Cell Application Development

by

Joseph Keith Hammer, Jr.

Master of Science in Computer Science

University of California, San Diego, 2007

Professor Scott Baden, Chair

The IBM Cell processor is a heterogeneous multi-core architecture designed to demonstrate exceptional levels of performance improvement for compute-intensive applications. The streamlined design of its Synergistic Processing Units (i.e. small local store, no cache, limited branch prediction, no dynamic instruction reordering) presents a new set of challenges for application developers as they are now required to explicitly control the flow of data amongst the processors.

Therefore, in this thesis we introduce a lightweight, flexible framework library called Cell Architecture Framework and Extensions (CAFE) to assist developers in taking advantage of this computational power without forcing a single programming model onto their applications. CAFE takes a more minimalistic approach than other frameworks by presenting low-level abstractions and utilities designed to help partition and transfer data between the cores. As a result, programmers can develop

applications using a reasonably high level interface, yet still retain explicit control over the flow of data. We believe that these characteristics are very important to developing high-performance applications in the Cell environment.

To motivate our design decisions, we provide an in-depth examination of where these aforementioned implementation challenges appear and discuss a range of countermeasures that may be successfully employed. We also present an assortment of example applications that each utilize CAFE in a different manner to help show its versatility.

# Chapter 1: Introduction

## 1.1 Architecture Trends and Revolutions

Undoubtedly, programmable computer architecture has come a long way since its beginnings in the mid-twentieth century. It is reasonable to say that for the past half-century we have been "living under the cover" of Moore's Law, which states that "the complexity for minimum component costs has increased at a rate of roughly a factor of two per year" and that "certainly over the short term this rate can be expected to continue, if not to increase." [Moore65]. This prediction has remained as a comfortable trend in computer hardware engineering, despite the fact that the speed of memory has not been able to match the pace.

However, although the size of transistors continues to shrink with each new generation, we have found ourselves in an interesting time when Moore's Law has started to fail due to some fundamental physical laws, such as power consumption and heat dissipation. These obstacles have pushed computer architects in a different direction: rather than using the next generation of transistors to increase the complexity of the processor in an effort to gain operational speed, architects have started to duplicate more simplified versions of processors across the chip in a stamp-like manner. This has given rise to the dual- and quad-core processors available on the conventional hardware market today.

Before this transition occurred, another revolution took place during the mid-90's: the advent of conventional graphics hardware accelerator. Graphics Processing Units (GPUs) are a type of coprocessor specialized for performing common operations

found in computer graphics: transformations, interpolations, etc. In particular, they are designed to take a chunk of data (from hereon also referred to as the "payload") and perform the same set of operations on each of its components. For instance, at a high-level, a simple three-dimensional object is composed of a mesh of triangles that define its geometric shape and a set of texture coordinates associated with each triangle vertex to define its appearance. When we want to place the model in a scene for visualization, we must apply a transformation to all of the vertices of the triangles in the mesh, followed by an interpolation of the texture using the corresponding texture coordinates across each triangle for the sake of rendering. Since there may be thousands (or millions!) of triangles for each object, it is easy to appreciate the benefits in having dedicated hardware support for such operations. It is also readily apparent that such operations are independent of each other (no triangle vertex transformation depends on the results of any other triangle vertex transformation, and so on…), and therefore can be computed in parallel.

As graphics hardware has matured, there has been a definitive movement in the direction of programmability and generality. It now presents us with an architecture that performs the aforementioned operations in parallel at a very high rate of throughput via a paradigm called *streaming*. Streaming (or more formally: stream processing) performs the same computation across the elements of a single payload. Until recently, the streaming components supported on the GPU were strictly graphics-specific, including dedicated vertex and texture memory buffers and operation kernels in the form of vertex and pixel shader programs. However, this meant that anyone who wanted to use the GPU simply as a coprocessor for performing

non-graphics-specific computations was forced to repackage and mask the data as if they were a vertex or texture buffer as well as reformulate computations as if they were vertex or pixel shaders. Such a task can actually be quite an undertaking and is somewhat analogous to asking someone to cram a square peg into a round hole—it can probably be done, but it will take a lot of unnecessary force and the end result isn't going to be pretty. As it turned out, after seeing the effectiveness of the GPU in the realm of graphics, other areas of computer science—especially the scientific computing community—began to take an interest in utilizing its power for their own purposes, producing a new demand for more generality in the native operations and pipeline. The hardware and research that stemmed from this movement has been labeled General-Purpose Computing on the GPU (GPGPU).

## 1.2 Parallelism

There are two types of parallelism exploited in modern computing: control-level parallelism and data-level parallelism. As we will see, both types of parallelism are useful and important in their own respects and each contributes to hardware and software efficiency in their own unique way.

### 1.2.1 Control-Level Parallelism

Control-level parallelism is the action of processing two different courses of execution at the same time. This is made possible on many different levels within a machine (hardware threads, hardware operations on different units) and applications (software threads). On a single processor, the façade of performing multiple tasks

simultaneously (or *multitasking*) is simulated by managing the amount of time each task is allotted to spend on the processor. This act of scheduling is typically performed by the operating system (or a user-level thread scheduler) to ensure that each task is given a chance to make progress. On multi-core processors, the operating system will usually split the task queue among the available processor cores and handle the scheduling on each core individually.

Control-level parallelism is also very important to utilizing the processor efficiently. The classic example is when an application requires a piece of data from a slower device (e.g. hard drive) or from across a network. Since many hundreds or thousands of cycles may pass before that data arrives, it would be more efficient to allow the processor to attend to the computational needs of other tasks in the meantime.

We can draw a correspondence between the CPU and control-level parallelism. This is the reason why conventional CPUs are designed to switch among different tasks (more formally known as *context-switching*) as efficiently as possible: during normal operation, the CPU must handle frequent context-switches amongst all the software applications running on the machine, each with its own program counter and application-specific data processing operations.

### 1.2.2 Data-Level Parallelism

As one may have suspected, data-level parallelism is the act of performing the same set of operations on different pieces of data simultaneously. In accordance with convention, we will refer to this set of operations as a *kernel*. What's more is that the

techniques of data parallelism can be applied to many different levels within the code: from the aforementioned kernel level even down to the level of individual instructions.

The best concrete example of this is type of parallelism is found in a Single Instruction Multiple Data (SIMD) instruction [Tommesani03]. For instance, consider how vector arithmetic finds the sum of two vectors *A* and *B*: we simply add each of the components independently:

$$(a_1 \quad a_2 \quad \cdots \quad a_n) + (b_1 \quad b_2 \quad \cdots \quad b_n) = (a_1 + b_1 \quad a_2 + b_2 \quad \cdots \quad a_n + b_n).$$

This is exactly the mechanics behind SIMD addition, and can be extended to a number of other computational and conditional logic operations. In the interest of multimedia applications, SIMD operations are conventionally performed on 128-bit quadwords at a time, as shown in Figure 1.1 below.

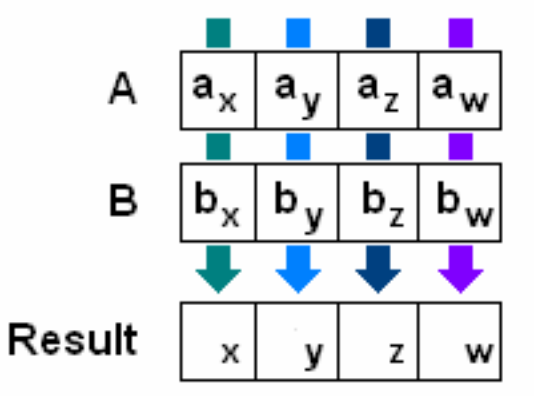



**Figure 1.1: Quadword SIMD Operation**

Data-level parallelism is at the heart of processing on the GPU since its purpose (at a low level) is to perform the same operation over a large set of data. In fact, we can also identify the multiple layers of data parallelism being performed: at the highest level we have multiple triangle meshes that make up our scene; at the

component (triangle vertex) processing level we have a list of independent transformations; and at the instruction level we have SIMD operations.

### 1.3 Next-Gen Architecture: IBM Cell

Although we have dedicated an entire chapter to the Cell architecture, we will briefly introduce it here enough to present the context of the work that follows.

In short, the Cell processor is currently on the bleeding edge of hardware technology. In terms of its architecture, it is a hybrid between the multi-core processors and the GPUs we introduced earlier, though assembled heterogeneously. Cell has nine separate cores: one for running the operating system and performing control-level parallel tasks (PPU), while the other eight conventionally serve as coprocessors for data parallelism though they can operate independently of the PPU's control (SPUs). As a whole, the Cell was designed around the streaming paradigm, though in a more general fashion as the eight coprocessors are not built on graphics-specific structures like a GPU.

The Cell processor was specifically designed to perform massive amounts of single-precision floating-point operations as quickly and efficiently as possible. Unlike a conventional CPU which is designed to perform reasonably well for a general mix of applications, the Cell is specialized to demonstrate excellent performance on a core set of target applications (including multimedia processing and compression, real-time simulation, and games), while exhibiting less-than-desired performance results for other types of applications (such as operating systems or database management software). As such, software packages from each of these core target application

categories require an enormous amount of computational power. Thus, the eight coprocessors within the Cell were designed to perform extremely fast single-precision floating-point operations in parallel.

However, although it makes for a very interesting configuration for a parallel machine, the Cell's architectural design promotes a few fundamental differences within code development practices from those found in conventional parallel computing say, with MPI [MPI03]. These differences include a rigid constraint on memory alignment and a limited remote local store. However, the foremost of these differences is the exposure of the memory hierarchy to the programmer, who is now expected to carefully manage *both* code and data across the coprocessors. In fact, developers can dynamically upload code to the SPUs. While this capability gives developers a great deal of control and the opportunity to achieve previously unattainable performance benchmarks, it also imposes a steep learning curve, which requires even the novice programmer to have a deep understanding and intimate knowledge of the underlying hardware.

In consequence, the work presented in this thesis shows the struggles and triumphs that have been required in an attempt to tame this beast, and presents a solution in the form of a set of libraries, which we refer to as a "framework", to ease the efforts of other aspiring Cell developers.

## 1.4 CAFE: Cell Architecture Framework and Extensions

Data partitioning and, more generally, data transfer preparation is a common problem in *all* non-trivial Cell applications. However, while the IBM-distributed software development kit (commonly referred to as the CellSDK) does an excellent job at providing low-level hooks into the hardware, it also makes the partitioning of data a meticulous and error-prone task, due to the level of intimacy required between the programmer and hardware (previously introduced in Section 1.3). It has come with our own experience that the mechanics of data partitioning can distract developers from spending their time on the more interesting areas of the application, such as scheduling, data flow, and vector processing, not to mention the ubiquitous issues behind the design and implementation of the algorithms and overall architecture of the code base.

We are not the only ones to have recognized this trend as both commercial and academic solutions have been developed to help alleviate this issue and offset the learning curve set forth by the Cell processor. Unfortunately, these solutions have also taken it upon themselves to provide their clients with a proxy pipeline that confines application development into a particular paradigm as opposed to allowing the programmers to retain control of the scheduling and data transfer methods utilized in order to accomplish their tasks. We believe that the power to manage the scheduling and transfer of data is particularly important in the area of high performance computing, and recognize that they should always be approached on an application-to-application basis since it is more often the case that *the programmer knows their data far better than any library ever could*.

In the light of this observation and others, we set out to create a minimalistic framework to assist the Cell development community without usurping the control over performance critical application components. The thesis that follows is a discussion of our work and a presentation of our framework libray: Cell Architecture Framework and Extensions (CAFE). Our main contributions include a module to automatically assess and partition data sets into suitable payload sizes for the Cell's SPUs, where the bulk of the computational load is resolved. We have also formalized and provided the structures required to establish a data transfer preparation paradigm for ferrying payloads to and from the SPUs without seizing control of the method of scheduling. Finally, due to the nature of our research lab, we designed the framework to ease the porting and development of the graphic and algorithmic processes employed by the Scalable City project (see Appendix A).

## 1.5 Conventions

Briefly, we would like to lay out a few conventions used throughout this thesis. First and foremost, we are primarily concerned with developers as opposed to application end-users. In fact, we refer to programmers who utilize our framework or the framework of others as *clients*. Consequently, we will use the words *developer*, *programmer*, and *client* interchangeably.

Additionally, in this thesis we will discuss various *design patterns* for Cell application development. These patterns are also referred to as *templates* and *programming models*. Furthermore, these patterns are usually focused around the

*pipeline* (read: flow of data) between the cores. As such, we will sometimes simply refer to the patterns by the pipelines they entail.

The term *kernel* is used throughout the thesis, yet is very context-dependent. For example, in Chapter 3 different APIs refer to their SPU programs as kernels, while in the image processing application of Chapter 6 kernel refers to the image filter. We have been careful to define this term as the context changes.

## 1.6 Thesis Outline

To conclude our introduction, we present an outline of the thesis to follow and highlight some of the important features of the road ahead.

First, we will introduce the Cell processor in detail and touch on some of the finer points required for developing non-trivial applications for it. While a complete overview of the hardware would require a more in-depth discussion, we believe that Chapter 2 provides sufficient detail for a solid understanding of the hardware and the capabilities that it provides.

Second, in Chapter 3 we will review prior and current work on APIs and languages. In particular, our discussion will revolve around how prior solutions help ease the steep learning curve that has been set by the complexity and exposure of the Cell architecture to the programmer.

Third, we will examine the initial work that was done to explore some of the limitations of the hardware as well as produce some baseline results in order to help motivate some of the design decisions made in CAFE. We believe this to be a very important chapter as it reveals the difficulties found in Cell development at the

implementation level. Furthermore, we will also present and evaluate various possible approaches to augmenting these difficulties. It is from this set of methodologies that we generate a foundation for our framework.

In Chapter 5, we will formally introduce the framework library and the various design decisions that lie behind its conception and implementation. In addition to a presentation of the structures and mechanisms available at a conceptual level, we also provide a pair of complete templates for utilizing CAFE in a Function-Offload manner.

Afterwards, Chapter 6 presents of a few implementation examples using our framework to build Cell applications. We have chosen these applications based on a number of factors including exhibiting the versatility of applications our framework constructs support.

Finally, we will discuss some future work that could be done to continue to extend the framework as well as further improve application development on the Cell processor.

## Chapter 2: Introducing the Cell Processor

We now provide a brief introduction to the IBM Cell Broadband Engine (or as it is more conventionally called: "Cell") hardware. This chapter is by no means comprehensive, but is sufficient to give the reader an appreciation for the capabilities of the system. Throughout, we will be sure to visit particular programming aspects which stem from the unique design decisions made for the Cell.

We would also like to note that unless specified otherwise the specs given below are for a single Cell processor only and scale within the Cell BladeServer, which has two Cells per blade and up to four blades in the system at UC San Diego.

The following specifics and details have been gathered from [Kahle05], [Gschwind00], [Hofstee05], the *IBM Cell Broadband Engine Architecture Guide* [IBM05], and the *IBM Cell Programming Handbook* [IBM06a].

### 2.1 The Cell Processor

The Cell is made up of two types of processors: a PowerPC front-end (Power Processing Unit: PPU) and eight synergistic vector processors (Synergistic Processing Units: SPUs). Each processor type requires a different, separate program to be written and compiled with different modified versions of `gcc` (or `g++`) provided in the CellSDK. It is interesting to note that the generated binaries can be run individually on their respective processors; programs for the PPU are just like any other application written for the PowerPC architecture, while the programs written for the SPUs, called "spulets", are designed to be simple and "encourage porting and incremental

refinement of legacy code on the SPU" [IBM spulet sample]. In our case, we will only discuss the instance when the two types of processors are interdependent.

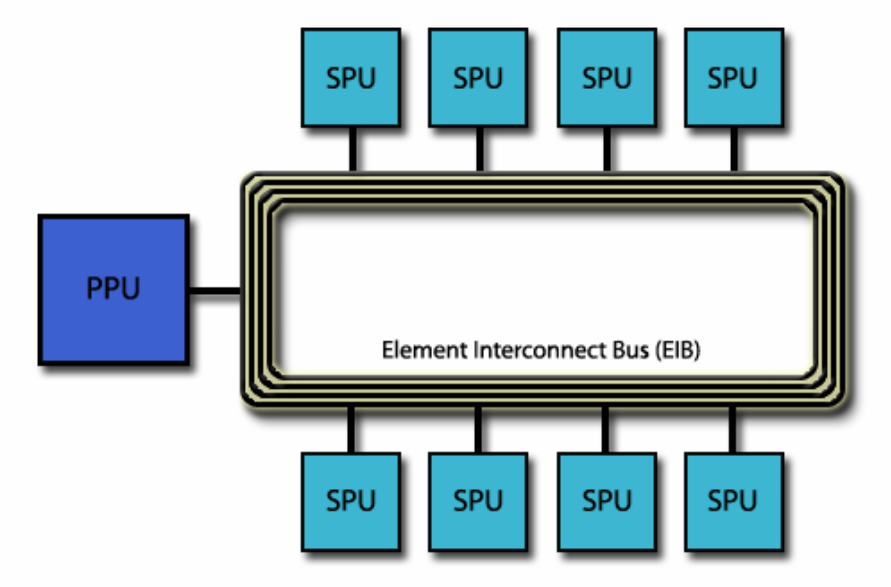



**Figure 2.1: Cell Processor Floorplan**

## 2.2 PPU: PowerPC Processing Unit

As mentioned above, the PPU is just like a conventional PowerPC. It is a 64-bit RISC processor, clocked at 3.2 GHz. It includes a VMX vector processing unit which supports the AltiVec SIMD vector instruction set extensions. It has 32 128-bit vector registers, separate instruction and data L1 caches (32 KB each), and a unified 512 KB L2 cache. There are also two available hardware threads and a traditional virtual memory subsystem.

The PPU's main purpose is to run the operating system, manage system resources, and serve as the control manager for the SPUs. In general, there is little difference (if any) in programming for this side of the Cell from conventional C/C++ application development practice.

## 2.3 SPU: Synergistic Processing Unit

Synergistic Processing Units are named for the mutual dependence they have with the PPU. In terms of architecture, the PPU is designed to handle control-intensive tasks and perform efficient task-switching, while the SPUs are designed to handle compute-intensive tasks. Essentially, this means that the SPUs rely on the PPU to run the application control-thread and feed them both code and data, whereas the PPU relies on the SPUs to perform most of the computational heavy-lifting[1].

### 2.3.1 Processor

The SPU is also a 64-bit RISC 3.2 GHz processor, but runs its own custom "vector only" instruction set. We say "vector only" because even scalar operations are loaded into vector registers and run using vector operations during execution. This translates to the following:

1. Insert scalar value into the preferred slot[2] of the vector register
2. Perform the operation using the corresponding vector operation
3. Extract the scalar result from the preferred slot of the vector register

As one might presume, this is a very important fact to keep in mind when programming for the SPU as scalar operations are not truly as atomic as they look from a code perspective.

The SPU has a unified register file containing 128 128-bit vector registers. By "unified" we mean that all supported data types (integer, single-precision floating-point, and even logical operators) use the same register file.

Also, the SPU is a dual-issue processor. This means that it has two separate pipelines, named *odd* and *even*, for executing instructions and can execute one instruction from each pipeline every cycle. All load and store instructions are processed in the odd pipeline, while all other operations are executed in the even pipeline. The pipeline can issue a single-precision floating-point operation every cycle with a latency of six cycles, while a double-precision floating-point operation can be issued every 7 cycles in addition to the same 6-cycle latency.

### 2.3.2 Local Store

One of the major design goals for the SPU was predictability. Essentially, the designers at STI[3] wanted a *deterministic operating environment* in which one can statically determine the performance of code. In other words, they wanted to establish a set of rules such that the running time of the same code would always be the same. While at first glance this may sound a bit silly, but consider the flow of data that goes on under the hood of a conventional system: when an instruction is to be executed, the data operands must be loaded into registers, but this data may currently reside anywhere in the memory hierarchy… L1, L2, L3 (if available), main memory, on-site storage, or maybe even storage located across a network. This implies that we may not know how long it could be before that data may be ready for the instruction to use. Certainly, in most cases the data is close to the top of the hierarchy thanks to spatial or temporal locality, or both, but the fact still remains that the time it takes for a piece of data to be loaded into a register is unknown and can only be narrowed down to a range of cycles depending upon where in the hierarchy the data currently resides.

This brings us back to the issue of predictability on the SPU. To attain the goal of creating a deterministic operating environment, the designers at STI wanted to avoid the irregular latency times built up by moving data up and down memory hierarchies. The solution was the aforementioned large register file in combination with a 256 KB SRAM local store (LS). The LS is allotted for both code and data, and can *always* be accessed in 6 cycles. Also, all loads and stores are made on a 16-byte aligned boundary without translation, paging, or protection.

We would like to make it clear that although it is a relatively small, fast memory close to the processor, the local store is *not* a cache, wherein pieces of data are brought in and swapped out under automatic hardware control. In fact, there is no cache anywhere on the SPU. This leads to the exciting and important observation that since there is no cache, there can be no cache misses!

One final detail must be added to complete our discussion of the SPU's characteristic of predictability: *in-order execution*. In the interest of speed, most conventional processors today have the ability to execute instructions out of their specified issue order (referred to as *out-of-order execution*), provided that the ordering does not affect the program correctness. This feature is, by design, *not* available on the SPU. Thus, we gain the ability to walk down the assembly code instruction by instruction and assign cycle times to each line for the entirety of the program. This allows for SPU binaries to be statically timed, providing Cell developers with a definitive guide to help them examine and streamline any stalls that may be present in their code.

### 2.3.3 Branch Prediction

While the predictability attribute of the SPU architecture sounds pretty nice, there is one final, yet important feature to discuss: branch prediction. There is *no* hardware branch predictor present on the SPU. In other words, all branch prediction is done at the software level. There is an 18-cycle latency for mispredicted branches. To counter this performance penalty, there are a couple of software mechanisms in place, including an inline branch-hint directive and a *select* intrinsic[4].

The inline branch-hint directive is exactly what it sounds like: in the code, the programmer can specify whether the branch should be predicted as taken or not taken (by default, all branches are predicted as not taken). The directive can be used for either static or dynamic branch prediction (see Section 1.6 of *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture* [IBM06e] for more details).

Given a bit pattern, the *select* intrinsic allows for individual bits to be selected from one vector or another. For example, given two vectors A and B, we can choose bits from A or B with a bit pattern P, where a 0 in P says to choose the bit from A and a 1 in P says to choose the bit from B.

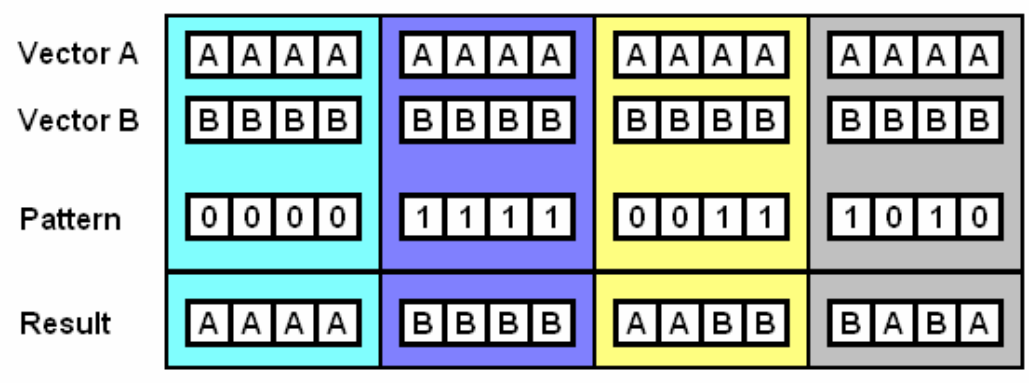



**Figure 2.2: SPU Select Intrinsic**

The significance of the select intrinsic from a programming perspective is that one can compute both routes of a branch and then select the result based on a pattern vector which can be created via one of the comparison intrinsics. This is often desirable since it puts the issue back in the SPU's dominant domain of raw computation.

## 2.4 Communication Mechanisms

Communication between the two types of processors is handled via three mechanisms: mailboxes, Direct Memory Access (DMA), and signal notification registers. It should also be noted that SPUs can communicate with other SPUs, which allows developers to employ a programming model called Chaining (see Section 2.6); however, in the interest of simplicity, here we will focus the discussion on communication between the PPU and SPU.

### 2.4.1 Mailboxes

Mailboxes are queues used to pass small (32-bit) messages back and forth between the PPU and an SPU. Each SPU has two types of queues: a pair of outbound (SPU to PPU) queues, and an inbound (PPU to SPU) queue. It should be noted that messages cannot be passed between two SPUs via mailboxes.

Both outbound mailbox queues are only a single message deep; one is for general purpose messages to the PPU and the other is used for interrupts. The PPU may poll these queues before making the blocking call to actually read from them. A practical use for this is when the SPU has finished processing its current batch of data

and needs to tell the PPU it is done; we certainly do not want to hold up any other tasks the PPU may need to do (data preparation, checking other SPU threads to see if they have finished, etc…), so we poll the SPU outbound mailbox, respond appropriately, and continue on. We will take a look at the other side of this technique in Chapter 4 when we discuss the SPU-driven pipeline.

The inbound queue can hold up to four messages at a time for the PPU to send to the SPU. However, if the PPU sends more than four messages to an SPU, the fourth message will continue to be overwritten by subsequent sends until the SPU performs a mailbox read. To avoid this undesired side-effect, one could have the PPU send the first four messages and then wait on the SPU to reply through the outbound mailbox that all the messages have been read, signaling the PPU that mailbox message sending may resume. This simple design pattern serves as a way to synchronize the two processors if the application requires such a sequence of short messages to be dispatched to the SPU.

Mailboxes are useful for sending addresses and action codes (e.g.: exit / continue, success / failure) as we shall see in Chapter 5. However, sending 32-bits at a time will not suffice for real-world data processing – that's where DMA comes in.

### 2.4.2 Direct Memory Access (DMA)

The main method of data transfer on the Cell processor is Direct Memory Access (DMA), which takes place on the Element Interconnect Bus (EIB). Briefly, the EIB (see Figure 2.1) is composed of four rings for transferring data at an internal

bandwidth of 96 bytes per cycle. Two of the rings are directed in the clockwise direction while the other two are counter-clockwise.

On the EIB, DMA requests are handled asynchronously, allowing for 128 simultaneous transfers between main storage and a SPU local store (shared across all eight SPUs). Each DMA transfer is invoked with a user-defined tag which can later be used to wait on the transfer's completion by reading the tag status. It is important to note that DMA requests can be made in batches by simply invoking multiple transfers with the same tag $t$; then, when the tag status for $t$ is read, the program will wait for all of the corresponding requests made with $t$ to complete. We will discuss further applications for tags in Chapter 4.

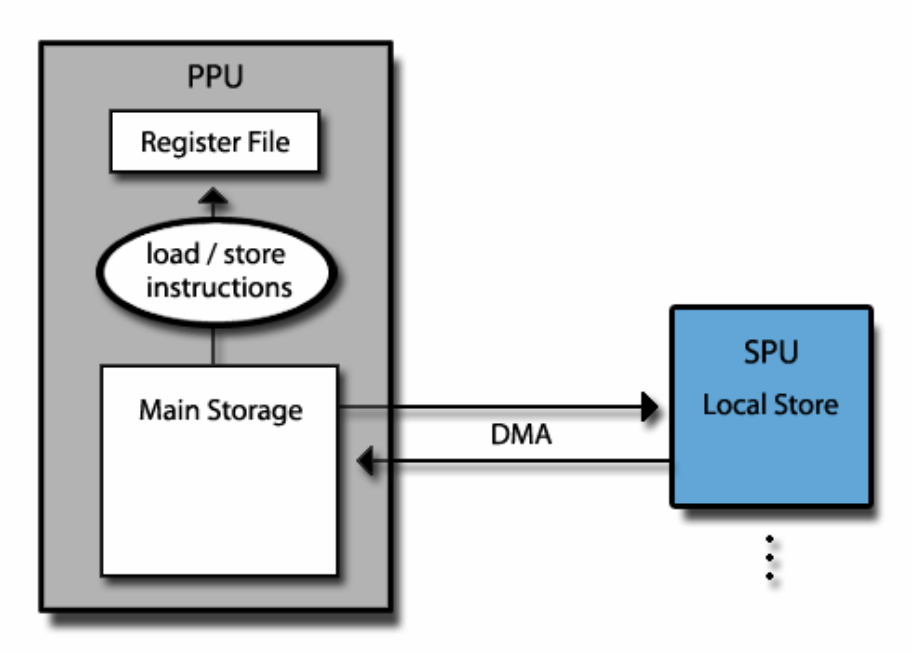



**Figure 2.3: Cell Memory and Communications Structure**

The SPUs are designed to input / output data in increments of 128-bytes up to 16 KB per request. This is also subject to the constraint that all memory DMAed to

and from the PPU must be 16-byte aligned, though the vendor recommends that transfer data be 128-byte aligned for best performance.

It is interesting to note that either processor type may invoke a DMA request from the other; however, the Memory Flow Controller (MFC) resides on the SPU so no matter where the request originated, all requests are actually made from the SPU under the hood (this is also why the queue names "inbound" and "outbound" are designated from the perspective of the SPU). Therefore, to attain peak performance it recommended that DMA requests are exclusively made from the SPU.

### 2.4.3 Signals

The SPU signal notification channels are kind of like another pair of inbound mailboxes. There are two channels, each for 32-bit signals. An important difference between notification channels and configuration mailboxes is that the former can be configured in a reduction (many-to-one) in addition to their single direct-line configuration (one-to-one).

We note that we have never employed the use of signal notification channels as mailboxes and DMA have been sufficient for the needs of all of our applications to date. Signals have been suggested as a means of implementing a barrier construct for synchronizing the execution of the SPUs (see IBM DeveloperWorks Forum: *Barrier* Post [IBM07]). However, a barrier can also be more efficiently implemented through the use of mailboxes, which is the method we chose to employ in our framework and use in the Chaining example discussed in Section 6.3.

## 2.5 Programming Models

There are a few different programming models described in the Cell Programming Handbook. The two most notable models are *Function-Offload* and *Chaining*. As we shall later discover, the Function-Offload model is the more prominent of the two, largely due to its simple structure and function. However, although the Function-Offload model serves as the basis for most of the program examples discussed in this thesis, we would like to emphasize the importance of Chaining and have dedicated an entire section on examining it usefulness and implementation in Section 6.3.

### 2.5.1 Function-Offload Programming Model

At its core, the Function-Offload programming model is analogous to a Remote Procedure Call [STI05]. It entails using the PPU for data preparation and SPU thread management, while the SPU is used for the data processing. Figure 2.4 shows this exchange of data between the two processors and an outline of their duties.

The Function-Offload paradigm is intuitive with respect to data parallel programs: simply create multiple SPU threads, each running the same SPU program, and partition the data such that each SPU has a portion of the workload. This way the data can easily be operated on in parallel.

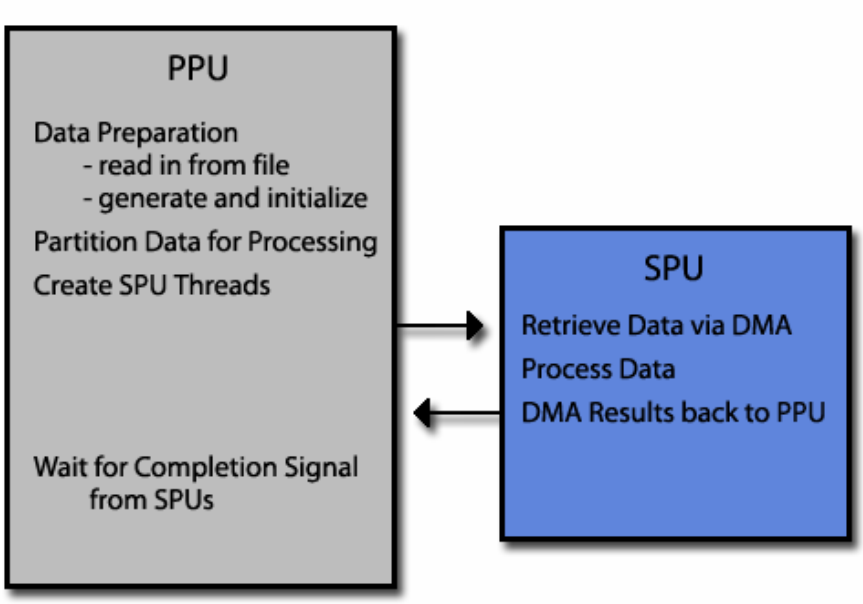



**Figure 2.4: Function-Offload Programming Model**

Certainly, the SPU binaries need not be the same across all eight SPU's. This means that we could setup half of the SPUs to do one operation and the other half to do another (or whatever configuration may be needed to best fit the data and requirements of the application). Implementing this technique allows us to employ task-level parallelism on top of the aforementioned data-level parallelism. To be sure, this flexibility is an extremely important characteristic of the Function-Offload programming model and should be exploited whenever appropriate.

### 2.5.2 Chaining Programming Model

The Chaining programming model is based on the idea of a firefighter bucket brigade. First, the PPU sends a chunk of data to a SPU, which performs some amount of processing on the data. Once complete, the SPU hands the data off to the next SPU

in line, which, in turn, performs its own operations. This continues down the chain until the processing is complete and the data is finally sent back to the PPU, as shown in Figure 2.5. Most assuredly, we can add the mechanics of pipelining into this procedure so that once the data has left the first SPU in the chain it can retrieve the next chunk of data to process in parallel.

This paradigm works best with data that must be processed in stages, such as applying filters for image processing. It can also be seen as an extension to the Function-Offload programming model where the chain in a Function-Offload setup has only one link.

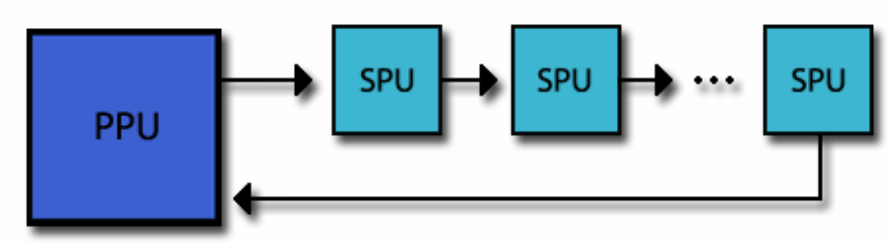



**Figure 2.5: Chaining Programming Model**

As with the Function-Offload programming model, we will have more to say about Chaining in Chapter 4 when we examine some of the fundamental operations inherent in every Cell application.

## 2.6 Summary

In this chapter we have introduced the Cell processor's two processor types: the PPU and the SPU, how they communicate through mailboxes, DMA, and signal notifications, and a pair of useful programming models for developing applications on

the Cell. We have also tried to point out important programming related aspects throughout the hardware discussion in hopes to give the reader a firm grounding for some of the higher-level details to come.

**Chapter 2 Notes:**

1. We acknowledge the fact that the SPUs can operate on their own (as "spulets") but for the purposes of this work we will only discuss the SPUs as helper coprocessors to the PPU.
2. The *preferred slot* refers to a 32-bit slot in a 128-bit vector register where results from reduction operations are stored. This is also the slot where scalar operations are performed.
3. STI refers to the Sony Toshiba IBM Cell Design Center in Austin, Texas, which was established in March of 2001.
4. An *intrinsic* is a lightweight function corresponding to a single or small set of assembly instructions performing a fundamental operation.

## Chapter 3: Related Work

While Cell is still a very new platform, it is amazing how many "solutions" have been presented to assist developers in the new challenges presented by the hardware architecture. In this chapter we will discuss several of these projects in hopes to set the playing field for the forthcoming discussion of our contribution. Additionally, we will study a pair of solutions that have been presented on the more general topic of programming stream processors to help establish some of the common ideas related to Cell in this area. Finally, we will examine some of the published performance benchmarks for Cell, as well as compare and contrast the designs and features of the framework solutions.

### 3.1 Stream Processing Solutions

A streaming processor independently executes a kernel over all elements in a given input stream and places results in a corresponding output stream. Examples of stream processors include conventional GPUs such as the [ATI R600] or [nVidia G80], the AMD Stream Processor [AMD Stream], and, in the light of its SPUs, Cell.

As alluded to earlier, stream processing can yield enormous boosts in program performance for particular application classes by exploiting data parallelism. This very same concept applies to the Cell SPUs and should be employed whenever possible.

In this section we will discuss a pair of related solutions designed to assist programmers in developing software applications to efficiently take advantage of the

powerful benefits offered by stream processors: the Brook programming language and the PeakStream Platform.

### 3.1.1 Brook Programming Language

Brook is an extension of the C programming language designed to assist developers in writing applications for stream processors. It was initially designed for the Merrimac Streaming Supercomputer Project at Stanford University. Since then, it has stemmed a descendant language called BrookGPU (from hereon just "Brook"), which allows developers to use the same language semantics to write stream programs for GPUs. For the purposes of our discussion, we are primarily interested in the GPU derivative as described in [Buck04] and [BrookGPU].

The Brook programming language presents the GPU as a general coprocessor rather than a graphics-specific one. Similar to other stream-oriented solutions, it was designed for computationally-intensive data-parallel applications with an emphasis on portability across consumer GPU platforms without sacrificing performance.

Brook defines programming constructs for *streams* and *kernels*. A *stream* is a collection of data to be processed independently in parallel. A *kernel* defines the scope in which streams may be operated on. More precisely, a kernel is a special function (designated by the `kernel` keyword) that internally loops over the input stream(s), performing the set of operations within the body of the function for each element. Brook also provides the mechanics for writing reduction operations to compute a single value from an input stream in a data-parallel fashion.

On the GPU, streams are stored as textures and kernels are compiled into pixel shaders. The Brook compiler is responsible for splitting the kernel into multiple passes if the number of outputs requested is greater than what the hardware supports (this varies depending upon the shader version supported by the particular GPU). This process maps onto the programmable graphics pipeline nicely since it is analogous to the method graphics programmers employ to produce composite effects (lighting is computed in one pass, blur is applied in another, etc.).

The Brook Runtime (BRT) software layer sits on top of OpenGL [OpenGL] and Direct3D [Direct3D] (the leading software interfaces for graphics programming) for both nVidia and ATI APIs, allowing it to utilize the benefits of GL or D3D directly in order to achieve the best possible performance. BRT also has a CPU backend, as well as a "close to metal" (CTM) backend [AMD CTM] to utilize the new initiative put forth from the ATI/AMD merger [Stokes06a]. Backends are chosen at runtime (either by hardware detection or user request) so there is no need to recompile Brook source code to switch.

### 3.1.2 PeakStream Platform

The PeakStream Platform stemmed out of the Brook project outlined in the previous section [Stokes06b]. It was targeted to serving the High-Performance Computing (HPC) community[1], with a standardized platform for developing computationally-intensive applications in C/C++ based on the stream programming model. Although the platform is no longer available in the commercial market[2], we

still would like to discuss the main ideas behind their feature set as they continue to remain quite relevant to our own interests in developing for the Cell.

The PeakStream Platform was a full product suite that included a compiler, virtual machine, system profiler, and a developer API with semantics similar to MATLAB [MATLAB] (in terms of its native support for matrix/vector operations) [Stokes07b]. Similar to other virtual machine based solutions, this implied that once compiled, the application could be run on any PeakStream supported platform.

Under the hood, the PeakStream virtual machine handled the threading and scheduling for the execution of applications on specific hardware configurations, such as multi-core CPUs and CPUs with available GPUs [Woo07]. Programmers did not need to be aware of which hardware they were running on—only the structures and mechanics presented to them through the PeakStream API. This meant that the PeakStream platform was essentially an architecture-agnostic standard for writing applications for stream processors.

Parallel kernels were created by the virtual machine at runtime, which allowed for configuration-specific optimizations to be performed before the kernels were sent down the pipe to the hardware cores. The company took care to emphasize the point that since "target processors may have widely divergent architectures, it is important to make kernel density and boundary decisions at runtime rather than hardcoding them at application design time" [PeakStream07].

The API included several data types, the foremost of which was the *Array* (akin to a matrix in MATLAB). It also included an extensive math library that contained routines ranging from basic arithmetic operations to matrix multiplications

and convolutions. Moreover, the available math functions were written to be far more accurate than those native to the GPU hardware—another key point appealing to the targeted High-Performance and Scientific Computing communities. For example, [PeakStream07] presents a plot of the relative error for the `exp()` functions from a GPU (unspecified) and the PeakStream VM; while the GPU implementation results in relative error approaching 1.0e-05 as the magnitude of the function input grows, the PeakStream implementation results remain grouped around 1.0e-07 throughout.

The PeakStream Platform certainly sounded like a serious solution to many of the current problems facing application developers of this new generation of stream processor architectures, especially in the areas of hardware abstraction and platform portability. Unfortunately for Cell developers, their emphasis was always more on exploiting the power of GPUs than hardware such as the Cell (undoubtedly due to their foundations in Brook).

### 3.2 Function-Offload Solutions for Cell

We will now turn our attention to the most prominent programming paradigm for Cell: the Function-Offload programming model. While there are other solutions which employ the Function-Offload model (some of which we will discuss in the next section), the Offload API and the RapidMind Development Platform proved to be the purest implementation examples of this paradigm. However, it is interesting to note that these frameworks provide two very different interfaces for presenting the same programming paradigm to their clients. As such, we will examine each of these in detail and compare their benefits and limitations.

### 3.2.1 Offload API

The Offload API was developed by the Parallel Programming Lab at the University of Illinois at Urbana-Champaign as a byproduct of porting the Charm++ runtime system to the Cell [Kunzman06a] [Kunzman06b]. Although Charm++ uses the Offload API to extend the portability of applications that utilize Charm++ to the Cell, the Offload API is independent of Charm++; therefore, we will limit our discussion purely to the Offload API.

In the Offload API, developers designate a *work request* to be executed on the SPU. A work request associates processing code with a set of three data buffers: read-only, read-write, and write-only. To ensure data-parallelism, a work request must be independent of the other work requests that may be currently executing on the SPUs. Work requests are asynchronously dispatched to an event-driven SPU runtime that handles the mechanics of transferring input data to the SPU, executing the processing code, and transferring the output data back to the PPU.

However, the Offload API operates under the following constraint: code must already be loaded onto the SPUs previous to the initialization of the Offload API. This implies that there is a major caveat to their paradigm: developers are forced to package all of their processing code into a single binary to be distributed across all of the SPUs, a characteristic that may be prohibitive to certain applications since, as we noted earlier, the 256 KB local store houses both code and data.

Another caveat, though not as acute as the first, is that the Offload API does not expose any methods for bringing more data to the SPU during the work request execution. This means that large operations must be serialized into several work

requests, which, in turn, invokes an unnecessary pair of data transfers in between the stages as intermediate data must be returned to the PPU only then to be DMAed back to the local store where it once was.

Finally, we note two other minor caveats. First, the Offload API, as its name suggests, does not support other programming paradigms, such as Chaining, which may prove to be more efficient for particular applications. Secondly, the Offload API does not allow the programmer to specify a particular SPU to run on. This is handled by the API automatically. Such knowledge is useful for tasks like sorting as well as running a non-homogeneous set of programs on the available SPUs.

In spite of these limitations, the Offload API provides a nice library solution that allows developers to easily push jobs onto the SPUs without attending to the matters of scheduling.

### 3.2.2 RapidMind Development Platform

The RapidMind Development Platform [RapidMind] is a commercial product developed by RapidMind Inc. (formerly Serious Hack Inc.). It grew out of an open-source research project from the University of Waterloo Computer Graphics Lab called *Sh*, a metaprogramming language that is embedded inside C++ for programming GPUs [libSh06]. Like Sh, code for the RapidMind Development Platform (from hereon just "RapidMind") is evaluated and compiled at runtime for the target platform (multi-core CPUs, GPUs, or Cell), but is far more general purpose.

With RapidMind, clients write a single source program as a single thread of execution. Then, within this source developers specify which parts of the code they

want RapidMind to split and execute on multi-core processors in a data-parallel manner through the use of the metalanguage constructs [Du Toit07] [RapidMind07a].

The RapidMind API supplies its own *value* and *array* types. RapidMind values are fixed-sized tuples of scalars; they can contain any of the C++ standard types (such as `int` or `float`) or any of the supported non-standard types (such as a half-precision floating-point). Values can come in different sizes, though those with one to four elements are the most common. As one might expect, a RapidMind array is simply a contiguous sequence of RapidMind values. By using RapidMind arrays, the developer informs the backend that all operations performed on the elements contained within can be considered independent of each other, which allows the dynamic compiler to exploit data-parallelism and split the data accordingly so that computations can be divided amongst the cores.

Also, in the RapidMind API kernel functions are called *programs*. RapidMind programs are actually objects that are executed like a function but make special use of the *value* and *array* types. When arrays are passed in as parameters to a program, it is analogous to calling the program on each of the array elements individually, which is an act of data-parallelism that can be exploited by the RapidMind backend without the need of any further direction by the programmer in terms of the data's alignment, partitioning, or transfer to the other cores for parallel processing.

To be sure, the main benefit to using RapidMind is that it allows the programmer to quickly gain the benefits of multi-core processors without having to explicitly deal with the issues of threading, message-passing, or data transfer. As a

close second, once the code is written, it is portable to any platform that RapidMind supports, including a wide range of multi-core CPUs, GPUs, and Cell-based systems.

Another notable aspect of developing with RapidMind is that there is no need for programmers to uproot their present development environment. All of the aforementioned functionality is wrapped in a library that is linked via the C++ compiler—no strings attached. Furthermore, the API supports incremental development by giving the programmer the freedom to choose which parts of the code will use RapidMind, which allows them to quickly parallelize "low-hanging fruit" components. These characteristics are very important for developers who wish to parallelize large projects or legacy code.

On the other hand, RapidMind is an industrial-strength commercial package with a deployment license starting at $1500, which places it out of the reach of a portion of the Cell development community [Hearst07]. While it has been stated on the Sh site that developers at RapidMind Inc. plan to release an open-source version in the likeness of the commercial one, it may be some time before this actually happens. In addition, RapidMind covers up the scheduling of the SPUs. Since the inner workings of the package are unavailable, it is unknown if the RapidMind backend is capable of employing different programming paradigms, such as Chaining, in situations when it may be more efficient to do so. Finally, like the Offload API, the ability to choose a specific SPU or group of SPUs to run on is not an option when using RapidMind, though, again, this may not be one of the programmer's primary concerns when it comes to writing or porting an application to Cell.

## 3.3 Alternative Programming Models for Cell

In this section, we present some of the more exotic solutions to programming applications on the Cell: the Sequoia programming language, the MultiCore Framework, and Cell Superscalar. These solutions are each based on their own unique programming paradigms different from the ones covered so far.

### 3.3.1 Sequoia Programming Language

The Sequoia group claims that when writing a high-performance application, the developer must have "non-trivial knowledge of the underlying machine's architecture" [Fatahalian06]. However, this requirement naturally discourages portability as there is always a trade-off between portability and performance. Sequoia responds to the emergence of so-called "exposed-communication" architectures where data movement is controlled in *software* between local and remote address spaces. This additional level of control that these architectures provide comes at the price of further complicating the design and implementation of software applications. As such, the Sequoia programming language [Sequoia] is intended to allow the developer to construct bandwidth-efficient parallel applications for exposed-communication architectures while maintaining portability.

The Sequoia language presents an abstract hierarchical memory model (also referred to as a "tree of memories" [Sequoia]) that hides the lower level architecture configuration, e.g. a multi-core chip, Cell blade, or even a cluster of uniprocessors. This differs from approaches in other parallel languages such as Unified Parallel C [UPC] and Titanium [Titanium] since they are primarily designed to facilitate the

movement of data “horizontally” across the processors of a parallel machine, while Sequoia’s model facilitates the movement of data “vertically” through the memory hierarchy. Consequently, the programmer is required to design their applications to map onto this hierarchy in a divide and conquer tree-like manner.

One of the main benefits to Sequoia is that the architecture-specific method of communication via DMA, MPI message passing, or even a cache prefetch instruction, is all hidden under the hood, yet exposed through a uniform interface. This ensures developers can still gain the benefits of explicit communication while producing a portable solution. However, it should be noted that only vertical communication is allowed throughout the memory hierarchy, and thus it is not possible to directly pass data between sibling nodes. In order to cater to a node cluster in which there is no physical hierarchy of memory, Sequoia provides programmers with the ability to virtualize levels within the memory hierarchy utilized by the software. Data stored within a virtual memory level will be distributed across the physical memories of its children and when communication takes place between a virtual parent and its physical child, the appropriate horizontal transfers are employed. All of these virtual mechanics are transparent to the programmer and are handled by the Sequoia compiler and runtime. As a result, memory level virtualization provides developers with the flexibility to design their software around a memory hierarchy that best fits the needs of the application without having to be concerned with the possibility that levels of this custom hierarchy may be absent in the target platform(s).

Sequoia’s core abstraction is the *task*, a pure function similar to a Brook kernel though without the syntactical sugar provided via a stream data type. Tasks are run in

parallel (or serially if not run on a parallel architecture), and mark the granularity of parallelism presented by the language. Each task is isolated into its own address space without the ability to communicate with other tasks. Hence, Sequoia exclusively handles data-parallelism through the Function-Offload programming model.

Tasks are written to be executed at a particular level within the memory hierarchy exposed by the language. As such, tasks are considered "abstract" because they are not actually bound to any memory until they are compiled and *specialized* for a particular architecture. The step of specialization is delineated through a script-like mapping specification separate from the source code. This approach separates the algorithm from its tunable parameters and is useful for hardware-specific application performance tuning.

With regards to Sequoia's Cell implementation, tasks at the leaf node level (equivalent to the local store level) are executed on the SPU, while non-leaf "inner" node tasks (equivalent to the level of main memory) are executed on the PPU. Due to the size constraints of the SPU, individual leaf node tasks are compiled as SPU overlays[3] and injected at runtime. The Sequoia runtime launches threads for all available SPUs upon initialization, planting a lightweight event-driven driver on each processor. These threads are kept running for the duration of the application (more on why in Chapter 4). From there, all Sequoia code is dynamically transmitted to the SPUs as each task is performed during program execution.

One of the primary concerns with Sequoia is that it is a solution in the form of a new language. This can be a risky path since it requires developers to decide if the benefits presented in the structure and semantics of the language outweigh the time

and effort that go along with learning a new language as well as the difficulties inherent with porting existing software to use it.

### 3.3.2 MultiCore Framework

The MultiCore Framework (MCF) is an API developed by Mercury Computer Systems Inc. for programming *n*-dimensional matrix computations on heterogeneous multi-core processors [Mercury06b; Bouzas06]. It is actually a component of a larger product called the MultiCore Plus SDK , which also includes a Scientific Algorithm Library (SAL), Trace Analysis Tool and Library (TATL), Image Processing Algorithm Library (PixL), and SPE Assembly Development Kit (SPEAD-K) [MCF].

MCF is packaged in the form of a library and is built upon the principles of the Function-Offload programming model. Within their Function-Offload Engine (FOE), they employ a "strip-mining" technique to partition the dataset (being sure to consider all the aforementioned alignment constraints) into SPU-sized chunks called *tiles*, which they can then push through the framework pipeline in parallel [Mercury06a]. In other words, tiles serve as the payloads for parallel tasks so that the data can be partitioned, processed, and reassembled in an efficient manner.

At a high level, MCF presents the developer with a *manager-worker* paradigm, called a *network*. Upon initialization, the manager (PPU) creates a number of worker (SPU) threads, each with a SPU runtime kernel to allow it to participate in the MCF network. Workers can also be grouped into *teams* during the initialization of the program to provide a coherent way of performing different tasks in parallel (as shown in Figure 3.1). After the teams have been established, the manager creates task queues

for each team and sends them on their way. It then follows that team members proceed to pop tasks off the queues and process them in parallel. Giving the client the flexibility to organize and designate teams of workers for various data-parallel stages within their application is a very nice feature since it provides the option of an additional level of task-parallelism within the mechanics of the framework.

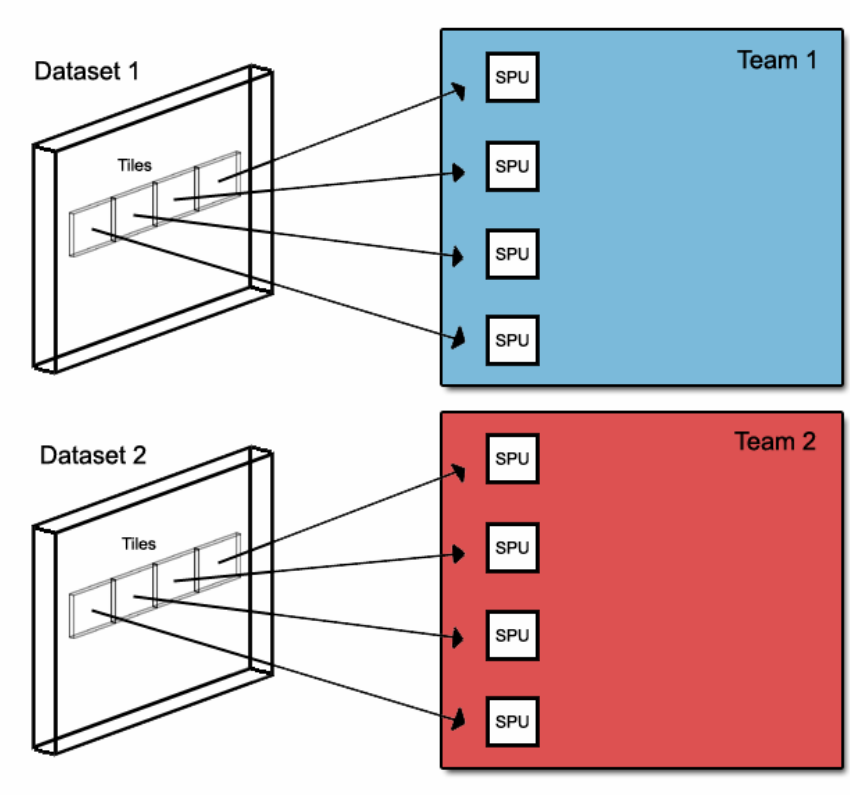



**Figure 3.1: MultiCore Framework Task-Parallelism via Teams**

During execution, the manager communicates with the workers via *input* and *output tile channels*. As each task is dispatched to a worker, metadata blocks indicate the appropriate tiles to retrieve as input and output buffers for the actual processing. While all of the communication handling is performed by the framework, the developer still has the responsibility of writing both the manager and worker source

code using their respective MCF APIs. In other words, MCF keeps the developer close to the hardware when it comes to performing the raw computation on the data, yet provides a convenient interface for partitioning and transferring the data between the processors.

However, one significant disadvantage of this framework is that it is targeted at applications looking to solve a specific kind of problem that is most naturally expressed as an *n*-dimensional matrix computation. While it is true that many problems can be broken down into small "tiles" from the original (such as a matrix multiply or the application a filter to an image), it may be difficult for some developers to formulate the solutions to their problems in this way.

### 3.3.3 Cell Superscalar

The Cell Superscalar (CellSs) project is the successor to the groundbreaking work explored in the GRID Superscalar environment by the Barcelona Supercomputing Center [Bellens06]. The primary goals have been to provide a simple, flexible programming model that can still take advantage of the performance benefits from a complex architecture like the Cell. CellSs is built on two principal components: the CellSs compiler (CSS) and a runtime library.

The CellSs compiler is based on the Nanos Mercurium compiler which was originally designed for OpenMP [Balart04]. It is a source-to-source translator that takes in an annotated C source file and produces a pair of source files, one for each Cell processor type (PPU and SPU). These source files can then be compiled by the respective compilers provided in the CellSDK. This approach enables developers to

write sequential code that is automatically parallelized by CSS in a functional manner. This approach differs from those previously discussed, which have been based on exploiting data-parallelism. CellSs simply relies on other resources, such as the Octopiler [Eichenberger06], to handle data parallelism and instead focuses on producing functionally parallel solutions from the sequential source.

In CellSs, developers add annotations to indicate candidate functions to be executed in parallel (on the SPU); however, like the `inline` keyword, these annotations are merely *suggestions* to the CellSs runtime. The runtime constructs a data dependency graph to schedule independent annotated functions to execute on different SPUs in parallel, allowing for *adaptive* functional parallelism. If there are dependencies found in the currently executing SPU tasks or if there are no free SPUs available on the system, then the function will be executed on the PPU allowing for the "sequential" application to continue to move forward. If a task is determined to be suitable for SPU execution, the runtime handles all of the data transfer management and scheduling.

In other words, CellSs provides developers with a tool to automatically parallelize a sequential application to run on Cell. While this is a phenomenal result in itself, the choices made by and the additional overhead from the runtime dependency graph may hinder an application from reaching peak performance. For example, from our own observation, it is frequently the case that when a particular task is run on the PPU it is much slower than running one or more similar tasks on the SPUs, which indicates that it may actually be faster for the application to simply wait for the SPUs

to finish their current task loads and then take on the task in question, rather than leaving it to be handled by the PPU.

## 3.4 Potential of Cell for Scientific Computing

Finally, we discuss work from the Lawrence Berkeley National Laboratory (LBNL) that analyzed the usefulness of the Cell processor in the Scientific Computing domain [Williams06]. The authors present a wide range of benchmarks relevant to the SC community, comparing Cell with a modified version of Cell ($Cell_+$), and other architectures such as the AMD Opteron, Intel Itanium2, and Cray X1E. Benchmark applications included a single- and double-precision general matrix multiply (GEMM), single- and double-precision sparse matrix-vector multiply (SpMV), stencil-based computations: a heat equation solver [Chombo] and a 3D hyperbolic PDE solver [Cactus], and single- and double-precision Fast Fourier Transform (FFT). In this section we will take a brief look at the architectural modifications and discuss some of the reported benchmarks.

### 3.4.1 Architectural Modifications

One of the main contributions of this paper emphasizes some architectural modifications to improve the efficiency of the double-precision hardware computations on the Cell without requiring a complete rewrite of the pipeline. This modified architecture is referred to as $Cell_+$. The modifications include lengthening the forwarding network to eliminate all but one of the cycle stalls. The authors claim that this improvement would allow for the chip to execute double-precision operations

every other cycle rather than the current 7 cycles (see Section 2.3.1). Simulated results boast that such changes would achieve as much as a 3.5*x* improvement in throughput of the current double-precision operations, without altering the other advantages offered by Cell (single-precision throughput, low power consumption, etc.).

### 3.4.2 Performance Benchmarks

By exploiting the deterministic environment of the SPU, the LBNL group developed a performance model for Cell to predict the total runtime for the computational kernels of their applications. Throughout the paper, this performance model is compared to actual runtimes from the Cell System Simulator and is proven to be quite accurate even for complex operations. For instance, when validating their model against IBM's implementation benchmarks for SGEMM, the performance model's prediction was within 2% of the actual measured performance (IBM: 201 GFlops; LBNL: 204 GFlops) [Williams06].

When juxtaposing their benchmarks for Cell and the aforementioned conventional processors, Cell clearly dominates in both single- and double-precision implementations. For example, their performance model estimated that Cell is 10*x* faster than the single-precision sparse matrix-vector multiply on the Itanium2, as well as 6*x* faster for double-precision. Also, their performance model estimates that Cell is more than 60*x* faster than the single-precision Chombo heat equation solver on the Opteron, and almost 13*x* faster for double-precision.

Overall, the benchmarks reported in [Williams06] make a strong case for the Cell processor and its significance to the Scientific Computing community.

## 3.5 Analysis and Insight

Before concluding this chapter, we would like to juxtapose the approaches of the aforementioned prior work to help put them into better perspective. In Figure 3.2 below, we have provided a chart summarizing the different framework approaches and their respective platforms.

| | Multicore CPU | PC Cluster | GPU | Stream Arch | Cell |
|---|---|---|---|---|---|
| Brook | | | X | X | |
| PeakStream | X | | X | X | |
| Offload API | | | | | X |
| RapidMind | X | | X | | X |
| Sequoia | | X | | | X |
| MultiCore Framework | | | | | X |
| Cell Superscalar | | | | | X |

**Figure 3.2: Framework Platform Comparison Chart**

### 3.5.1 Portable Development Solutions

To begin, if we consider portability as an absolute requirement for application development, our choices are narrowed down to PeakStream, RapidMind, and the Sequoia programming language.

Of these three, Sequoia provides the most fine-grain control but also requires the largest change in environment as it is a completely different language and programming paradigm. Additionally, Sequoia calls for developers to make algorithmic changes to their applications (or design them accordingly) if they are not already formulated in a divide and conquer tree pattern.

On the other hand, both PeakStream and RapidMind support incremental development, requiring only an adjustment of data types and code semantics at the points of desired parallelization. Another benefit that PeakStream and RapidMind

have over Sequoia is their portability to more conventional hardware configurations such as multi-core CPUs and GPU-assisted architectures, as opposed to PC node clusters[4] which are normally only available at institutions and centers of research.

### 3.5.2 Solutions for Cell Development

Now, if we turn our attention to considering each of these platforms as solutions for Cell development (discarding any possible portability requirements), our choices include: the Offload API, RapidMind, Sequoia, MultiCore Framework, and Cell Superscalar. We have compiled a chart to compare the parallelism and programming paradigms available through these frameworks in Figure 3.3.

| | Task Parallelism | Data Parallelism | Function-Offload | Chaining |
|---|---|---|---|---|
| Offload API | X | X | X | |
| RapidMind | ? | X | X | |
| Sequoia | X | X | X | |
| MultiCore Framework | X | X | X | |
| Cell Superscalar | X | | X | |

**Figure 3.3: Framework Parallelism and Paradigm Comparison Chart**

Firstly, the Offload API is the most limited of the aforementioned solutions due to its static code restriction and its one-to-one correspondence between a DMA payload and a work request task. However, the Offload API is available absolutely free of charge and the cost of integration is relatively small in comparison to the rest of the choices. Furthermore, the Offload API has no extra cost for dynamic compilation or code interpretation through a virtual machine. Therefore, in essence it keeps the developer close to the hardware yet presents a useful and intuitive abstraction of the data transfer pipeline.

As a Cell development platform, RapidMind still offers the convenience of incremental development but is considerably more heavyweight than the Offload API. Surely, RapidMind's strength comes from its architecture-agnostic API, through which the developer simply designates which portions of the code should be parallelized and permits RapidMind to handle the rest—memory alignment, DMA transfers, and even vectorization. However, this approach presents the developer with the complete opposite standpoint of the Offload API's intimacy with the hardware. With RapidMind there is no direct control over what tasks are presently executing in the SPUs at any given time—however, it is understandable that such knowledge may not be a defining concern for programmers and their applications and therefore may be an appropriate trade-off for the ease of development.

When considering Sequoia as a solution for Cell, the issue of language is still a significant concern. Nevertheless, in terms of its memory model, Sequoia presents a straightforward mapping of the memory hierarchy where inner nodes map to the PPU and leaf nodes map to the SPU as expected. This model is available to the programmer through the language, making Sequoia a viable prospect for high-performance computing applications wherein such control is imperative.

Although the MultiCore Framework has the constraint of predominantly catering to *n*-dimensional matrix computations, the abstractions presented to the developer are very intuitive and cover the most involved components to Cell application development: data partitioning and payload transfer. Furthermore, the additional support for explicit task-parallelism through the *team* metaphor is an important advantage over its commercial competitors.

Similar to RapidMind, Cell Superscalar abstracts away the architecture of the Cell and presents the developer with the task of designating which components of the program should be run in parallel on the SPUs without the need to write any communication or SPU-specific code. However, as we noted earlier, the additional overhead produced from constructing and maintaining the runtime dependency graph may hinder an application from reaching peak performance.

### 3.5.3 Framework Performance

In this section, we cite some relevant performance information concerning the aforementioned frameworks. However, we note that results for some of the solutions are few (or non-existent), including the Offload API's contribution to the NAMD project [NAMD].

RapidMind boasts that their dynamically compiled code can often outperform raw C or hand-tuned assembly code [Du Toit07]. Some of their results include a real-time Quaternion Julia Set Renderer capable of achieving over 40 frames per second, a real-time ray tracer (no exact figures given), and real-time crowd simulation (no exact figures given) [RapidMind07b]. These case studies and more (for comparative multi-core CPU and GPU-assisted examples) can be found on the Case Studies page of their corporate website.

The Sequoia paper outlines seven different benchmark applications which are "competitive with existing implementations of similar algorithms" [Fatahalian06]. They cite a result of 80.6 GFlops for their matrix multiply on a single Cell processor, and 160.7 GFlops when run on both processors on the blade, showing the scalability

of throughput. Furthermore, they report a 5% increase in performance over the leading GPU implementation of a fuzzy protein string matching application.

Mercury compares algorithm implementations utilizing their MultiCore Framework on the Cell against the "best available" implementations (Mercury or third party) on other platforms, such as 1.0 GHz Freescale 744x, 2.0 GHz PowerPC 970, 2.4 GHz Opteron, and 3.6 GHz P4 Xeon, in *Algorithm Performance on the Cell Broadband Engine Processor* [Mercury06a]. For their single-precision 64K Complex FFT, Mercury reports that they can achieve 90.8 GFlops of throughput (time between completions of successive FFTs while streaming), which is a huge increase over the second-ranking 6.07 GFlops by the Xeon (the other platforms clock in at around 3 GFlops). In other words, MCF on Cell offers a 15*x* speedup over the leading general-purpose processor. Similar results are cited for their symmetric image filter implementations.

### 3.5.4 Final Thoughts

In the light of performance reports like "Cell technology offers one to two orders-of-magnitude improvement in performance per processor and performance per Watt" [Mercury06b], we pose the following question: if Cell is so fast, what is holding it back from being fully embraced into more mainstream venues?

The answer is a matter of software. Simply put, Cell is not the easiest platform to develop for. Surely, the concepts and hardware details (which will be further discussed in Chapter 4) become more natural to programmers over time, but even then there still exists the additional layer of managing the flow of data across the cores.

Consequently, although the cited gains in performance are impressive, the weighty challenges present during software development can deter programmers from harnessing this power. For instance, if it takes a programmer a couple of hours to implement an application that takes a day to execute on a conventional processor, but takes a week to write the same application that runs in 20 minutes on the Cell, are they really saving time? However, if it is possible to gain even a portion of the computational power of the Cell for a fraction of this development time, then wouldn't the answer to our question be a resounding yes? This is exactly the reasoning behind the development of the frameworks and languages discussed in this chapter.

## 3.6 Summary

In this chapter, we discussed and compared several approaches to alleviate the challenges brought forth in developing applications for multi-core streaming architectures like the Cell. Initially, we examined the Brook programming language and the PeakStream Platform, designed to serve as portable solutions for stream processing application development on GPU-assisted systems.

Next, we investigated a simple library for developing applications under the Function-Offload programming model in the Offload API. The Offload API offers a straightforward interface, through an abstraction called a work request, to automatically handle the underlying details of transferring data to and from the SPUs. However, it is constrained by the fact that it cannot be used to transfer data payloads during the processing of a work request, which implies that large datasets must be serialized and incur additional overhead of duplicated data transfers.

Afterwards, we examined the RapidMind Development Platform, a commercial solution for developing applications for multi-core, GPU-assisted, and Cell architectures while maintaining portability across all three. RapidMind presents an architecture-agnostic interface that allows the programmer to designate which portions of code should be parallelized and offloaded to other cores in the system. However, RapidMind's high-level interface does not encourage developers to explicitly take control of the data transfer, forcing them to rely upon the internal decisions of the library to perform this performance critical step.

The Sequoia programming language presents the developer with the semantics for controlling the flow of data up and down the memory hierarchy. Sequoia requires applications to be implemented in a divide and conquer manner that ultimately forms what they refer to as a "tree of memories". Each level in this tree represents a level in the memory hierarchy. Furthermore, if a level within the tree does not have a corresponding physical memory, then the memory level is virtualized and its contents are distributed across the physical memories of its children, allowing programmers to design applications towards a particular memory hierarchy without being constrained only to architectures with a matching configuration. However, in order to take advantage of this level of control, developers must take on the additional task of learning a new language (plus environment setup, etc.), as well as reformulate their application to a divide and conquer tree if it does not already adhere to this pattern.

Next, we discussed the Multi-Core Framework which caters to assisting developers implement *n*-dimensional matrix computations on the Cell. In addition to this data transfer paradigm, MCF is supplemented by a set of companion libraries for

performing common operations from the mathematics and scientific disciplines. As in RapidMind, developers are forced to rely upon the internals of the library to employ the method of data transfer. Also, while *n*-dimensional matrix computations are indeed useful, it may seem unnatural for some applications to be restructured and implemented in this paradigm.

The final framework we examined was Cell Superscalar, which takes an approach that allows developers to write an annotated single source application that is automatically parallelized by their source-to-source compiler in a functional manner (as opposed to being data-parallelized). During execution, the CellSs runtime library constructs a dependency graph to choose which functions (suggested by programmer annotation in the original source) are suitable to be executed on the SPUs depending upon their availability (otherwise the task will be scheduled to execute on the PPU). However, the choices made by the runtime may schedule tasks in a suboptimal manner by scheduling a task to the PPU rather than waiting for the availability of the SPUs.

Finally, we concluded this chapter with a discussion of the benchmarks and architectural modifications put forth in the "Potential of Cell for Scientific Computing" paper, wherein the authors reported several measurements where Cell drastically outperformed the best implementations cited on conventional processors. The authors also claim that the latency of performing double-precision floating-point operations can be improved by lengthening the forwarding network to eliminate all but one of the cycle stalls. In accordance with this observation, their simulated results show as much as a 3.5*x* improvement in throughput when compared with the existing Cell hardware.

**Chapter 3 Notes**

1. PeakStream later stated that they had hoped to extend their reach into other domains such as gaming and image processing [Stokes07b].
2. PeakStream Inc. was bought by Google less than a year after its official launch in mid-September 2006 [Stokes07a].
3. Overlays are the mechanism by which code can be dynamically inserted and run on the SPU. They do not require a new SPU thread to be spawned with a different SPU binary, and thus save on the SPU thread startup time (see Section 4.1). This transaction of code is managed by an Overlay Manager as described in the IBM CellSDK SPU Overlay example.
4. At the time of this writing, Sequoia only supports the Cell and PC cluster platforms. Multi-core chips were labeled as an item of future work in [Fatahalian06] and may be supported in the near future.

## Chapter 4: Cell Microbenchmarks

In the Fall of 2006, I was given the opportunity to carry out the ground-breaking work for the gamelab[1] on the Cell BladeServer. During the rest of the quarter, I spent a bit of time investigating various aspects of the platform, including writing and running a few low-level benchmarks in an effort to help familiarize myself with the strengths and weaknesses of the hardware. It was based on these findings that I realized that a lightweight framework library would be an invaluable aid not only to the gamelab and the Scalable City project, but to the Cell development community in general. For this reason, we will briefly examine the results of these microbenchmarks as well as discuss a few of the lessons learned in order to help motivate some of the design decisions for the CAFE framework detailed in the following chapter.

Before proceeding, we would like to note that all of the results presented in this chapter (as well as the rest of the thesis) were measured on the IBM QS20 Cell BladeServer at UC San Diego. Also, unless otherwise specified, all of the reported measurements were taken while utilizing only a single Cell processor on one of the blades[2]. Finally, all of the microbenchmarks presented in this chapter were implemented using only the mechanics provided in the CellSDK; i.e. without the assistance of any support libraries (CAFE or other).

## 4.1 SPU Threads

One of the primary concepts to understand in Cell development is that the SPUs are treated as a file system in Linux and are accessed via the use of threads. Upon the creation of a SPU thread, the following actions take place under the hood:

1. the system loads the program image into the SPU local store
2. the program state is initialized
3. the SPU program is started
4. the SPU thread ID is returned to the caller

During this time, the calling thread on the PPU will be blocked until it receives the pointer to the thread ID, signaling that the SPU thread is underway.

SPU threads run to completion, as opposed to being context-switched in and out of focus by a scheduler. This is an important characteristic because it means that the program will not be interrupted and has ownership over all of the SPU's resources during normal execution.

### 4.1.1 SPU Thread Startup Time

As it turns out, SPU threads can take a considerable amount of time to start up. We say "considerable" because when compared to the time it takes to DMA a payload to the SPU for processing, the amount of time to spawn a SPU thread is relatively large (see Section 4.2.1).

To overcome the startup overhead of spawning a SPU thread, we need to maximize the utilization of the operations made available via that thread. Thus, it is best to keep the SPU binaries resident for as long as possible. Moreover, considering

that SPU threads run to completion, SPU programs should be written in an event-driven fashion and employ a *message-pump* paradigm similar to the one outlined in Figure 4.1 below. We will further explore and utilize this event-driven pipeline in later chapters.

```
Loop forever:
{
   Wait for signal from PPU
   If signal = EXIT:
      break

   Retrieve data
   Process data
   Send back results
}
```

**Figure 4.1: SPU Event-Driven Pipeline Pseudocode**

One may recall that we examined a more extreme example that made use of these principles implemented in the Offload API (see Section 3.2.1), where only a single SPU binary could be used for the application and event signals were used to choose different operation routines for data processing. Similarly, these same observations motivated the design of CAFE's DMA helper routines; however, our library does not impose such restrictions. We will revisit this in the next chapter.

Next, we will present the results obtained from a microbenchmark for analyzing SPU thread startup times to support the previous remarks.

### 4.1.2 SPU Thread Startup Timing Results

We begin by measuring the time it takes to spawn a SPU thread. Our microbenchmark uses a minimalist SPU program (it simply returns 0), which only profiles the time of the call to create the SPU thread (*not* the complete thread runtime).

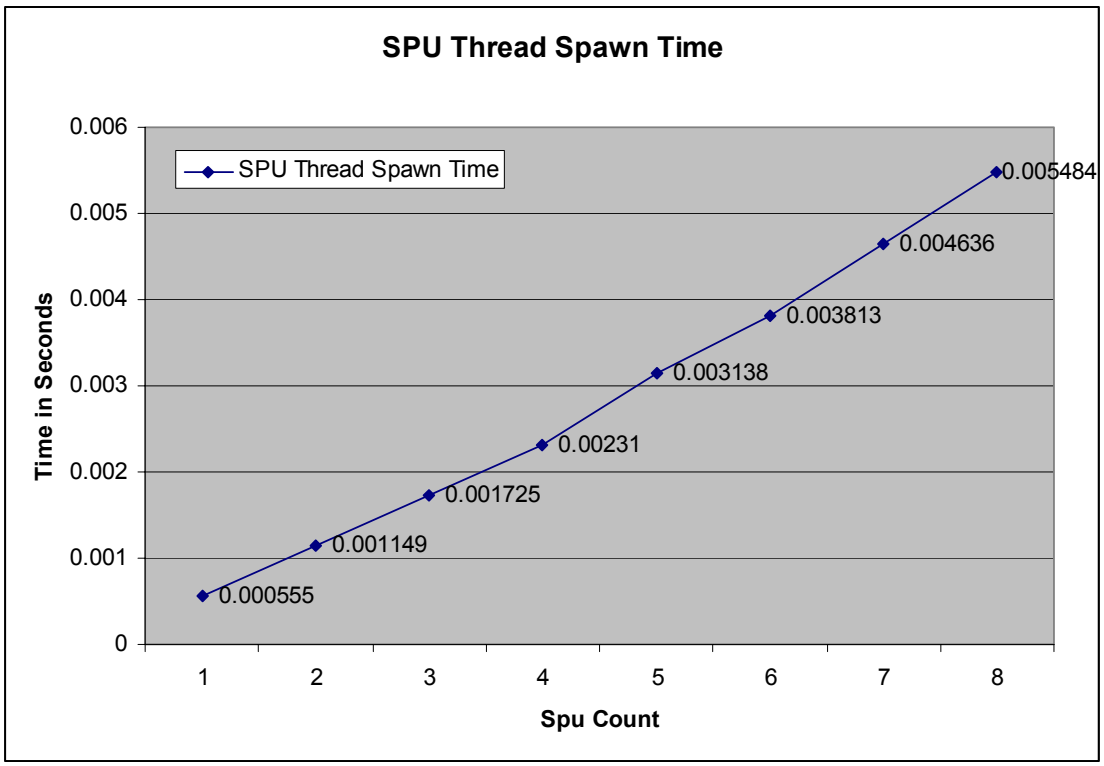



**Figure 4.2: SPU Thread Startup Time for Single Cell Processor**

The results in Figure 4.2 above show that it takes roughly half a millisecond to create a SPU thread and return the thread ID to the PPU. Furthermore, we also see that this time roughly scales with the thread count in a linear fashion up to the eight SPUs on the processor as we might have expected.

## 4.2 DMA Test

Most certainly, the Cell's power stems from the utilization of the SPUs for data processing. However, this power is of no value unless the processor can efficiently move the data into and out of the SPU local store. As revealed in Chapter 2, this is primarily accomplished via Direct Memory Access provided by the Memory Flow Controller. Since data transfer is such a vital component to Cell application development, we believed it was imperative to investigate the performance of DMA and pinpoint any particulars or caveats that may exist. As such, we next explore the cost of DMA.

### 4.2.1 DMA Timings

The first issue we would like address is the question of how long it takes to ferry a payload between processors. To answer this question, we implemented a small program to measure the time required to transfer data from the PPU to the SPU as well as from the SPU to the PPU. Figure 4.3 below presents the gathered results for various payload sizes from 16 KB to 192 KB (at increments of 16 KB) and compares the measurements for both directions of transfer. For this benchmark, we note that all of the results presented were collected from a single SPU and no data processing was performed on the transferred data. Also, all of the DMA transfers for both directions were invoked by the SPU to ensure that no extra overhead was incurred from any extraneous communication between PPU and the MFC on the SPU (see Section 2.4.2).

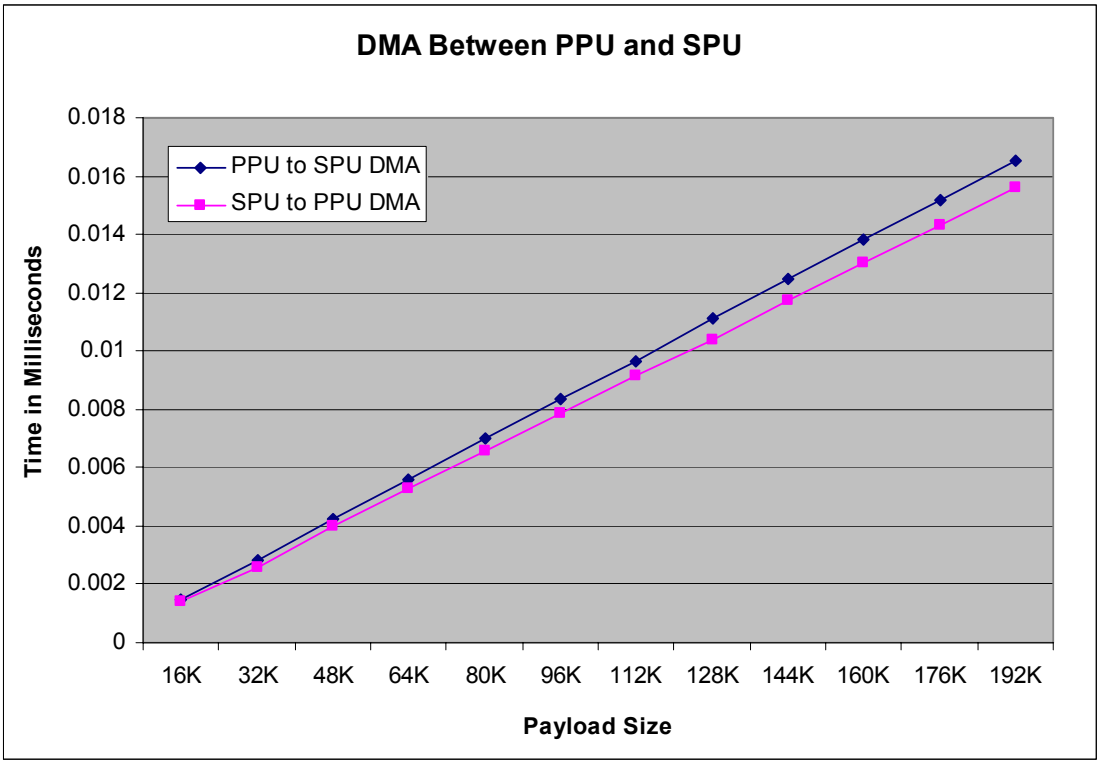



**Figure 4.3: DMA Transfer Time between the PPU and SPU**

Figure 4.3 shows the results for DMA transfers between the PPU and SPU in both directions. Take care to note that the times are labeled in milliseconds (as opposed to the SPU thread spawn times which were labeled in seconds). As such, it is important to realize that these benchmarks all fall well below the observed SPU thread startup time of half a millisecond from Section 4.1.2. Thus, these results provide the appropriate grounding for our previous assertion that spawning a SPU thread is a relatively expensive operation. This implies that it is best to switch out SPU binaries as few times as possible during application execution.

Along these same lines of advice, it should become clearer why the Chaining programming model from Section 2.5.2 is such an appealing paradigm for particular

applications. If we can load up different SPU binaries to handle different stages of the data processing pipeline, we will rarely, if ever, have to create a new SPU thread after initialization. What's more, we can save the extra DMA transfers back to PPU which would be present in a pure Function-Offload application by sending the data straight to the next SPU in the chain. We examine an example application that employs the Chaining paradigm later in Section 6.3.

Also, an interesting observation is that the DMA time required to send data back to the PPU is smaller than the time required for the SPU to receive data from the PPU. This was a peculiar find and we ended up re-running the benchmarks to ensure this observation was not simply a result of noise in the timings, yet the timings between the two directions stayed true to their original rankings (as shown).

### 4.2.2 DMA Method Comparison Test

Now that we have put the time for DMA in perspective, we next explore the mechanics of communication further by looking at a couple different methods of executing DMA requests.

First, we could write our communication code such that it is performed sequentially (see Figure 4.4a); that is, after each request we can read the tag status to ensure that the requested DMA transfer made with the current tag has completed. In other words, we could set up our pipeline to wait for each piece of the data payload to arrive before making the next request or moving on to begin the data processing. This is a reasonable manner in which to structure the SPU program's data transfer stage,

especially if data validation is required or there is a need to dynamically change what data is to be retrieved based on the results from previous transfers.

**Synchronous**

```
while(remainingData > 0)
{
   ComputeBatchSize()

   for all batches
   {
      Post DMA request
      Read tag status
   }

   remainingData -= totalBatchSize
}
```

**Asynchronous**

```
while(remainingData > 0)
{
   ComputeBatchSize()

   for all batches
   {
      Post DMA request
   }
   Read tag status

   remainingData -= totalBatchSize
}
```

**Figure 4.4: Synchronous vs. Asynchronous DMA Methods**

However, recall that DMA requests can be made *asynchronously* (see Section 2.4.2). This characteristic allows us to handle the transfer of data in a more "rapid-fire" manner. Recall that this can be accomplished by sending out all of the requests for the data we want to transfer with the same tag. Thereafter, we can wait on all of these requests with a single call to read the tag status (see Figure 4.4b). This way we do not have to stop the request pipeline as we wait on each individual transfer, rather we simply have to wait on the group as a whole.

It should come as no surprise that the asynchronous method is faster than the sequential one. Figure 4.5 confirms our intuition as it compares the data transfer time for both methods across various payload sizes ranging from 16 KB to 192 KB (uniformly spaced at intervals of 16 KB). We chose not to exceed 200 KB as it is nearing the capacity of the local store which also must house the code.

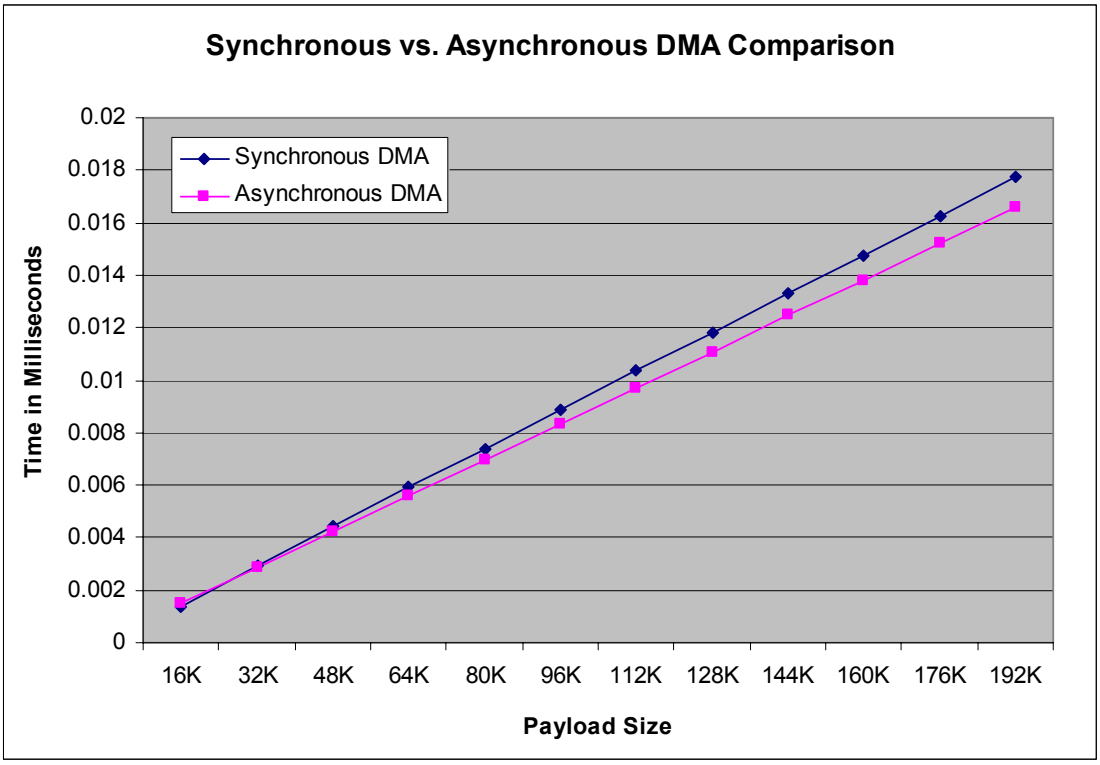



**Figure 4.5: Synchronous vs. Asynchronous DMA Transfer Comparison**

The plot above shows a steady increase in the difference between the times of the transfer methods while ferrying data from the PPU to the SPU, which is to be expected since with each additional 16 KB batch there is a corresponding synchronization wait. In the case of 192 KB, the asynchronous method holds almost a 7% margin of improvement in performance over the sequential method.

In accordance with these results, we recommend employing the asynchronous transfer method whenever possible. However, it should be noted that if an application happens to require the need for sequential data transfer (i.e. for verification purposes), it is comforting to know that the penalty for employing this method is not unbearable.

### 4.2.3 Final Recommendation

In practice, we recommend making asynchronous DMA requests in batches of 16 KB in size for transferring large payloads to and from the SPUs. More formally, any payload can be partitioned into two phases: the *batch* phase and the *final* phase. The batch phase simply dispatches as many 16 KB DMA requests that can fit into the total payload size without overflowing (think integer division), while the final phase makes a DMA request smaller than 16 KB to retrieve any remaining data in the payload after the batches have run their course (think modulus).

An example of this partitioning technique is shown for a payload size of 60 KB in Figure 4.6 below. In accordance with the technique above, the payload is partitioned in the following manner: $\underbrace{3 \times 16}_{batch} + \underbrace{12}_{final} = 60$. In other words, to fulfill this request we should dispatch three batches of 16 KB and a final batch of 12 KB.

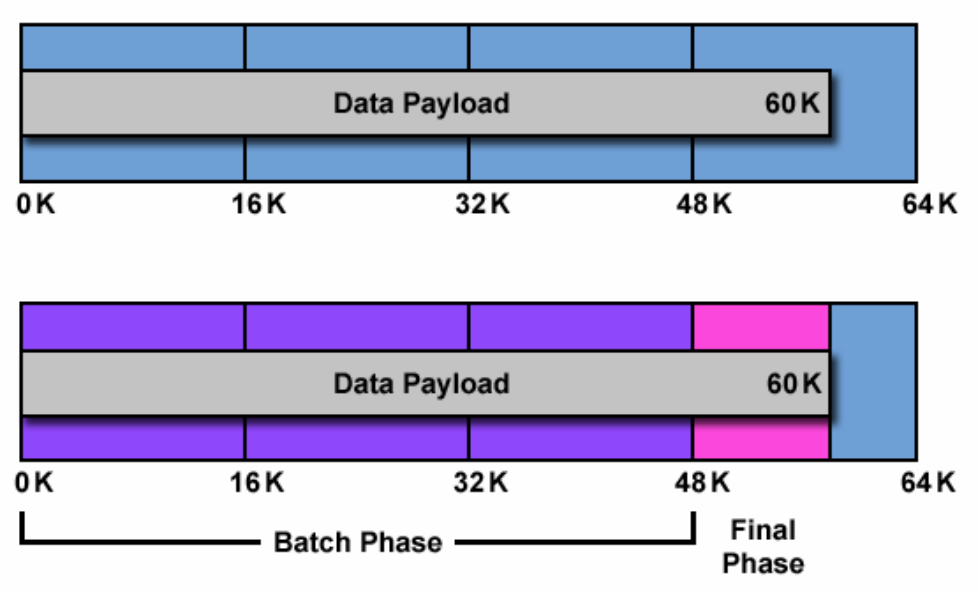



**Figure 4.6: DMA Transfer Split into Batch Phase and Final Phase**

This pattern began to appear throughout our application source code so frequently that we decided it should be generalized and refactored into a utility function. Hence, this is the method we follow for invoking DMA transfers in our framework implementation.

## 4.3 A Few Words on DMA

One of the biggest hurdles to tackle when first starting Cell development is the aspect of communication through the use of DMA. This is primarily because there are many fine-grained details to pay attention to as a result of the exposed architecture. Therefore, we would like to take a closer look at the mechanics of DMA to discuss some those details which have chronically been the source of many of our own implementation headaches.

### 4.3.1 Memory Alignment

First and foremost, it has often been the case that improper memory alignment *within the data itself* is from where most of our own troubles have stemmed. Aligning the data upon allocation is only the *first* step towards creating a DMA-compliant data buffer. When placing a DMA request, the addresses being submitted must be at least 16-byte aligned (128-byte aligned for peak performance), otherwise the program will render the infamous `Bus Error` message and abort. Surely, if the data is simple and happens to be granular enough to always split on these alignment boundaries after the initial aligned allocation, there is no need for concern. However, real-world data is

usually not this simple and happens to come readily sliced in nice 16-byte aligned chunks without undergoing some sort of alternative packing transformation.

One approach is to pad the data that will be transferred via DMA to be a multiple of 16 bytes. Unfortunately this can be challenging in applications where data structures are meticulously packed for the sake of memory and computational efficiency. In such cases, another solution is required; however, it can be the quickest and least design-intrusive in cases where these rigid restrictions are not present.

Another, more robust approach is to rearrange the data. Surely, if one is doing this strictly for the sake of DMA, the reengineering of carefully packed data could be a poor solution, yet it does point us in a favorable direction. As we alluded to earlier, the Cell performs *very well* when processing data parallel operations in a streaming fashion. Thus, if the data can be repacked into a set of stream buffers, the issue of alignment is partially alleviated as the substructures in the arrays have become more fine-grain and (hopefully!) easier to partition on a 16-byte alignment boundary (see Figure 4.7). Nevertheless, there is still no guarantee that this approach will work with every data structure without taking special care.

Indeed, the conditioning of data for transfer to the SPUs is a difficult problem and is undoubtedly best approached on an application-to-application basis. Nevertheless, its importance has prompted us to bring the issue to forefront of our reader's attention in hopes to steer them in the right direction. We also note that our framework is inherently partial to the streams method owing to the amenability of graphics processing to the decomposition of scene and image data into streams.

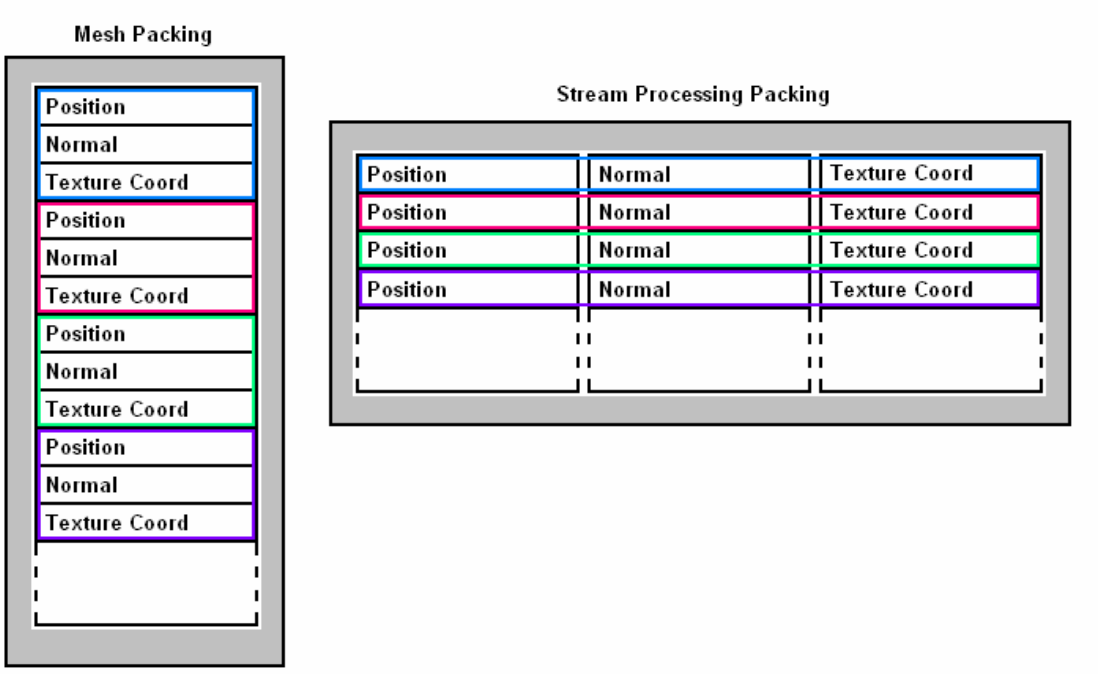



**Figure 4.7: Vertex Packing: Mesh vs. Stream Processing**

### 4.3.2 Data Transfer Size

The second issue we wish to address is that of DMA data transfer size. Individual DMA transfers are constrained to be no longer that 16 KB. On the other end, the minimum payload for a DMA transfer is 128 bytes. Then again, what has not been emphasized is the requirement for the size of a DMA payload to also be a *multiple* of 128 bytes.

In accordance with the first suggestion of resolving the memory alignment constraint, all data structures could be padded to be a multiple of 128 bytes in size. Fortunately, 16 KB is a multiple of 128 bytes so by solving the alignment requirement through padding the structure to be 16 KB aligned, the issue of payload size is automatically resolved.

However, what about the case of employing the streaming approach for small data structures (good examples being either control blocks or metadata structures)? Our solution is to perform a "round-up divide" and simply DMA the extra bytes. There are a couple of reasons for this. First off, transferring an extra 1-127 bytes is not going to affect performance horrendously; for the most part, most transferred data is handled within the batch phase and this extra memory is not as significant as the scale of the total payload transfer as a whole. Secondly, in order to reuse the structure after the payload has arrived, a data type cast is required to properly interpret the bits as the structure they once were; hence, these extra bits will not affect the integrity of the original data provided that the cast is performed at the correct starting address. Therefore, we have found that this approach is a viable solution to the matter of gathering the appropriate data transfer size.

## 4.4 Summary

In this chapter, we examined some of our early observations in hopes to lay down a foundation for some of the design decisions made in our framework discussed in the next chapter.

First, we observed that the startup time for a SPU thread is roughly a half of a millisecond. We also presented benchmarks for data transfers in both directions between the PPU and SPU which showed that the cost of DMA is small and confirmed that the DMA transfer time for a payload is much smaller than the aforementioned SPU startup time. This led us to the conclusion that it is best to keep the SPU programs resident for as long as possible.

Afterwards, we established a method of ferrying a payload to and from the SPU by designating a pair of stages: the *batch* phase and the *final* phase. The batch phase contains DMA transfers of 16 KB in size, while the final phase contains the single DMA transaction to transfer the remaining data in the buffer. Our final recommendation for performing data transfers on Cell is to use this dual-phase method and invoke the DMA requests asynchronously to wait on the group of requests as a whole and avoid any unnecessary waiting.

Finally, we discussed a few important details concerning the conditioning of a dataset for DMA transfer. In terms of data memory alignment, we first suggested additional padding to ensure that arrays of structures remain 16-byte aligned. We also suggested the repacking of structure components into streams, since the breaking of the agglomeration into finer-grained components is more likely to be split on 16-byte boundaries. Lastly, we addressed the issue of DMA transfer size and ultimately recommended the transfer of the additional data beyond the size of the actual structure since it does not affect the integrity of the real data, yet still allows for a DMA-compliant size for the transfer.

**Chapter 4 Notes**

1. The Experimental Game Lab (EGL; or just "gamelab") is one of Professor Sheldon Brown's labs at UCSD, located in the Center for Research in Computing and the Arts (CRCA) at the California Institute of Telecommunications and Information Technologies (Calit2). The gamelab is geared to the creation of new forms of art that utilize, extend, and connect technologies of computer gaming and scientific visualization.
2. As it turns out, by calling the function to create a SPU thread developers are given no guarantee that the thread is going to be created on the same processor as the PPU executing the application in a Cell blade environment. Therefore, we ensured that all of our SPU threads are kept local by using the `numactl` command to bind the processors and memory to a single Cell node [Kleen04].

# Chapter 5: CAFE: Cell Architecture Framework and Extensions

We now present our framework: Cell Architecture Framework and Extensions (CAFE), for developing stream processing applications on the Cell.

To begin, we will outline the goals and motivations we had set forth for ourselves during the preliminary design phase in developing our framework. We will then follow this by a conceptual overview of the structures and mechanisms provided by the library and discuss the positive repercussions our design echoes in using the framework implementation in our functional tests. After the mechanics have been introduced individually, we present a pair of example programming paradigms and pipelines that utilize these mechanisms to show CAFE's simplicity and versatility for general Cell application development. Additionally, we will walk through both a pseudocode template and an example program for each of these programming paradigms to give our readers an idea of how the API corresponds to the concepts introduced earlier.

## 5.1 Goals and Motivation

Our primary goal in developing CAFE was to create a minimalistic framework to assist developers in taking advantage of the computational power provided by the Cell's SPUs without forcing a particular programming model onto their applications. For instance, an application may be more naturally or optimally implemented in another programming paradigm, such as Chaining, as opposed to the Function-Offload model. In the light of these characteristics, we believe that CAFE is a unique solution

to this problem even amongst all of the frameworks and development platforms previously presented in Chapter 3.

First and foremost, CAFE is *lightweight*. We wanted a framework that was lean and learnable, so that other programmers would be able to quickly pick up and understand the structures and functional mechanics it offered. This made us very cautious about our decisions concerning what we found to be appropriate to include as a part of the CAFE library, all while withstanding the temptation to create a "kitchen-sink" solution, bloated with rarely applicable features. Therefore, we implemented several applications along the way to reconcile which components were absolutely necessary to cater to general Cell application development.

Second, CAFE is *flexible*. We wanted to design a framework that did not take over the application structure or flow. In particular, we wanted to ensure that the developer was free to implement their own scheduling routine and use the programming model best suited for their application rather than being required to use the one provided. Also, it is possible that an application may further benefit from a custom buffering technique over a standard double-buffering implementation. By only offering specific programming models or employing them in particular ways, developers may be denied the performance gains and implementation flexibility important—or perhaps critical—to their applications. Thus, we felt it was imperative to ensure that our framework offered the same levels of development freedom presented by the IBM CellSDK and no less.

Third, CAFE is *non-intrusive*. One of the original requirements for our framework was to make it easy to port a sizable legacy code base to the Cell (see

Appendix A). For this reason, CAFE does not require the client's code base to replace its own structures with custom data types in order to properly function. Instead, we decided that an intermediary metadata type was a better solution to the issue. Furthermore, CAFE does not require any additional setup (such as a change in compiler or development environment), nor does it confine a developer to putting their code in particular places while the framework maintains overall control of the application. In other words, we believe that the library code should live in the application's world—not the other way around.

Finally, CAFE keeps the developer *close to the hardware*. CAFE is, by design, low-level and without a runtime library (as was the case with many of the aforementioned solutions from Chapter 3). By striving to keep the developer close to the hardware, CAFE does not deny the programmer the ability to tune their applications in order to achieve the best possible performance. Also, a predetermined runtime library results in the loss of SPU local storage space for unused features on not just one, but across all the processors.

In consequence, these conclusions led us to design our framework to revolve around a flexible data-splitting solution with an *open-pipeline*. By open-pipeline, we mean to say that clients have the option of not being committed to the pipeline of ferrying data to and from the SPU for its entirety. In other words, our data transfer pipeline is divided into distinct stages which can either successively run their course or be replaced by the developer to suit their own purposes, ultimately presenting clients with the freedom to control the data flow and scheduling of the application through the use of payload-level abstractions. Most importantly, this also allows developers to

engineer their applications to best fit their own needs without being weighed down by the details of the mechanical requirements of the system.

## 5.2 Framework Mechanics

In this section, we will discuss the structures and mechanics featured in our framework. It is our plan to introduce these here at a relatively high level so that the concepts are not clouded by the details of their implementation and usage. We will show the lower-level details of using the framework to build Cell applications in the next chapter.

### 5.2.1 MemoryRegion

CAFE defines a primitive data structure called a `MemoryRegion`. The `MemoryRegion` is a clean way to pass around a pointer to data along with its size (in bytes). Primarily, it serves as a wrapper that allows our framework to operate on user-defined types and data without requiring them to integrate the `MemoryRegion` as a fundamental type into their code. We like to think of the `MemoryRegion` as a bridge between the developer's application code and the functionality provided in our framework. In other words, the `MemoryRegion` is a *low-impact* data structure, meaning that clients still own their data and are not required to package it into finer-grained data types for the library's convenience. This attribute of non-ownership is also why we refer to the `MemoryRegion` as *metadata type*, since it only holds higher-level information about some other actual data that it does not physically own.

In a way, the `MemoryRegion` is CAFE's version of a stream or array structure from the other solutions presented in Chapter 3, though it does not actually hold the data itself. To be absolutely clear, the memory that the `MemoryRegion` wraps is volatile, which means that it is the client's responsibility to ensure that the integrity of this data is not compromised (i.e. accidentally modified or freed) during its use. We believe that this attribute of non-ownership is more empowering than troublesome since it allows the framework to operate without duplicating the entire dataset, which could be a very expensive operation in terms of both time and memory.

### 5.2.2 `CafeJobs` and Data-Partitioning

CAFE is built upon the foundation of developing task-oriented applications. In CAFE these tasks are called *jobs*. A job is a unit of work performed on the SPU for a given payload. This payload normally comes from a larger dataset that must be partitioned into a number of individual SPU-sized chunks. Partitioning the original dataset is a necessary yet error-prone chore required for all non-trivial Cell applications (recall the numerous constraints outlined in Section 4.3). As a result, we implemented and thoroughly tested a general data-partitioning routine encapsulated in what we call the `CafeJobSplitter`.

The high-level goal behind CAFE's data-partitioning strategy is to exchange a large dataset for a queue of SPU-sized payloads, called `CafeJob`s (see Figure 5.1). Each `CafeJob` is defined as a contiguous section of some data that can fit within a user-defined size value no larger than the available memory on the SPU[1]. Like the `MemoryRegion`, the `CafeJob` is a metadata structure that actually only holds

information about where in the original dataset it starts and ends, as well as the number of chunks (16 KB or less) that are required to DMA the `CafeJob` from one processor to another in its entirety. However, also like the `MemoryRegion`, this means that the integrity of the original data buffer must remain pure throughout its usage during the related `CafeJob` execution; upholding this guarantee is the responsibility of the programmer since it makes little sense for the library to assume ownership of the client's data for the sake of communication.

Figure 5.1 depicts the process of generating the `CafeJobQueue`. It starts with the client's data wrapped in a `MemoryRegion` which is passed to the `CafeJobSplitter` which then performs the partitioning of the data (by reference) into `CafeJob`s.

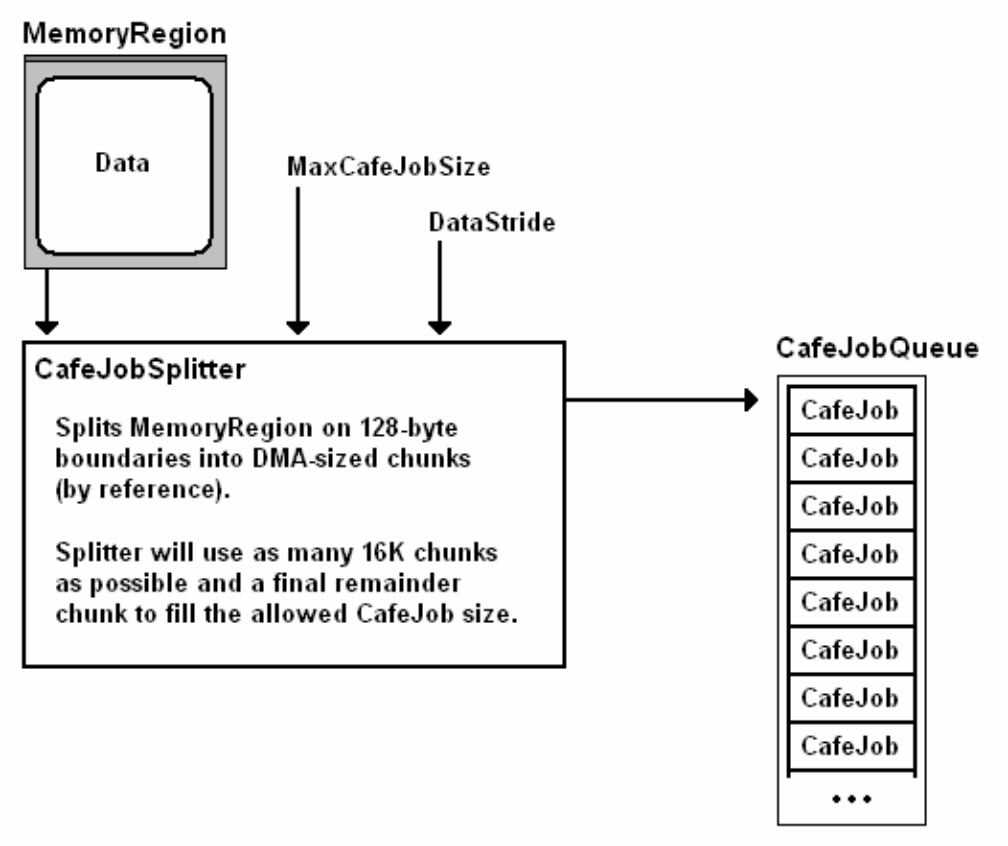



**Figure 5.1: CAFE Data-Partitioning Pipeline: Generating the `CafeJobQueue`**

Another benefit that stems from non-ownership of the data is that of reuse. As long as the data is not moved in memory or freed, the `CafeJob`s that point to it are still valid, and thus the `CafeJobQueue` may be reused for subsequent tasks, potentially saving on additional partitionings for the same data. Furthermore, in the case when the input and output buffers of the `CafeJob` are the same size, non-ownership also allows developers to reuse the memory of the input buffer as the output, producing the effect of a direct modification of the input data.

Above all else, we chose this simple representation because it gives developers the freedom to choose how to ferry their data to and from the SPU. If the data is already packaged into a single buffer, the developer may only need one job queue with a maximum `CafeJob` size approximately equal to: 256 KB minus the size of the SPU binary[2]. What's more, the developer may need to pull data from different buffers simultaneously to perform the task computations (recall the scenario presented in the stream processing packing from Figure 4.7); thus, in the same manner, it is easy to compute a partitioning of the available space that may be used by each of these buffers by simply dividing this maximum `CafeJob` size by the buffer count. Furthermore, the programmer is also given the freedom to choose when they want to initialize and dispatch their `CafeJob`s; i.e. some may want to precompute their `CafeJob`s and fire them off at a later time within their program, while others may need to fire them off in sporadic batches, etc.

However, we have overlooked an important detail: objects inside of data buffers are not always DMA-compliant (as was indicated in the final notes on DMA at the end of Chapter 4). In other words, while it is generally the case that buffers are

created to be aligned on a 16- or 128-byte boundary, the objects inside are not guaranteed to follow in lockstep. This becomes a problem when more than one `CafeJob` must be created to cover the size of the buffer... where do we start this next `CafeJob`? After all, we do not want to end up transferring half of an object at the tail end of one `CafeJob` and the latter half at the beginning of the next `CafeJob`—the results would be disastrous! Hence, `CafeJob`s come equipped with pointers to both the start of the data to DMA (properly aligned) and the start of the actual user data (may or may not be aligned)[3]. This is possible because in addition to requiring a maximum `CafeJob` size, our splitter also requires a *stride*, representative of the size of the object type contained within the buffer[4].

Internally, the `CafeJobSplitter` uses the stride to ensure that no structure within the buffer straddles a partition. For example, if the max `CafeJob` size was set at 50 KB and we had a buffer filled with four objects each 15 KB in size, we would not want the `CafeJobSplitter` to create two jobs where one is the full 50 KB and the other is the remaining 10 KB since it splits the final object between the two jobs. A correct partitioning may be found by using the stride to help guide the split locations; using a stride of 15 KB, the `CafeJobSplitter` will create two jobs where the first contains three of the objects (a total of 45 KB) while the other is the remaining object (a total of 15 KB). Here we note that while this result is in fact a correct partitioning, it not load-balanced. Due to the fact that we wanted the developer to have the opportunity to create and utilize multiple `CafeJobQueue`s to serve as the input buffers for a single SPU task, the `CafeJobSplitter` does not attempt to load-balance the data within the `CafeJob`s.

### 5.2.3 DMA Manager

Once the data has passed through the data splitter and the `CafeJobQueue`(s) have been established, the data is ready for transfer. At this point, the client may ferry the jobs to and from the SPU themselves using the mechanisms already provided by the CellSDK, or they can use the `CafeJobDmaManager`. This is one of those distinct points in our pipeline where the developer can choose whether they want to use the provisions of the framework or substitute in their own implementation.

In the case of the `CafeJobDmaManager`, DMA requests are invoked from the SPUs to transfer over the entire `CafeJob`, placing them in a buffer (designated by a `MemoryRegion`) as they arrive. To accommodate multi-buffering, the DMA requests are made in one call and a final tag sync (see Section 4.2.2) is made in a subsequent one. This allows for data transfers to be layered behind data processing in a simple and intuitive manner. Also, inside the `CafeJobDmaManager`, we employ the batch and final phase method introduced in Section 4.2.2 when dispatching the DMA requests.

If the developer knows that the scheduling of their data should be performed in manner other than the batch and final phase method to better meet the needs of their application, then they may implement their own data transfer scheduler for the DMA pipeline, yet still take advantage of the aforementioned job partitioning functionality. The structure of a `CafeJob` provides sufficient information for the developer to properly DMA the chunks (16 KB or less) and ascertain where the actual data begins and ends. Therefore, with this information it is possible to write a custom DMA scheduler to transfer `CafeJob`s to and from the SPU if so desired.

## 5.3 Programming Paradigms Using CAFE

At this point, we would like to present a pair of exemplary programming models that utilize the aforementioned structures and functionality provided by the CAFE library. For each model, we will provide a high-level overview, accompanied by a pseudocode implementation, and discuss its respective strengths and caveats.

### 5.3.1 Batch-Sync Programming Model

Recall from Chapter 4 when we found that in order to achieve optimal performance it is best to leave the SPU threads running with the same binary for as long as possible. Additionally, we determined that streaming data on a small scale can be expensive. Therefore, in an effort to find a suitable solution, we decided to stream our data at the job level, which led us to the following paradigm.

The *Batch-Sync* programming model is an event-driven paradigm in which individual jobs are deployed to the SPUs one batch at a time in a purely Function-Offload manner. In this method, the PPU is explicitly responsible for managing the workload for each SPU by maintaining the state of the coprocessors as they make their way through the `CafeJobQueue`. In other words, the PPU must determine how many `CafeJob`s are left in the corresponding `CafeJobQueue` and how many of those jobs can be issued to the available SPUs. We call this resultant set of `CafeJob`s which are all sent out simultaneously a *batch*. Although in Figure 5.2 (below) we show a batch of eight, it should be clear that a batch may be composed of any number of `CafeJob`s depending upon the number of SPUs the client wishes to dedicate to processing. To

be sure, CAFE allows clients the freedom to run several different batches simultaneously, even across the SPUs of multiple Cell processors.

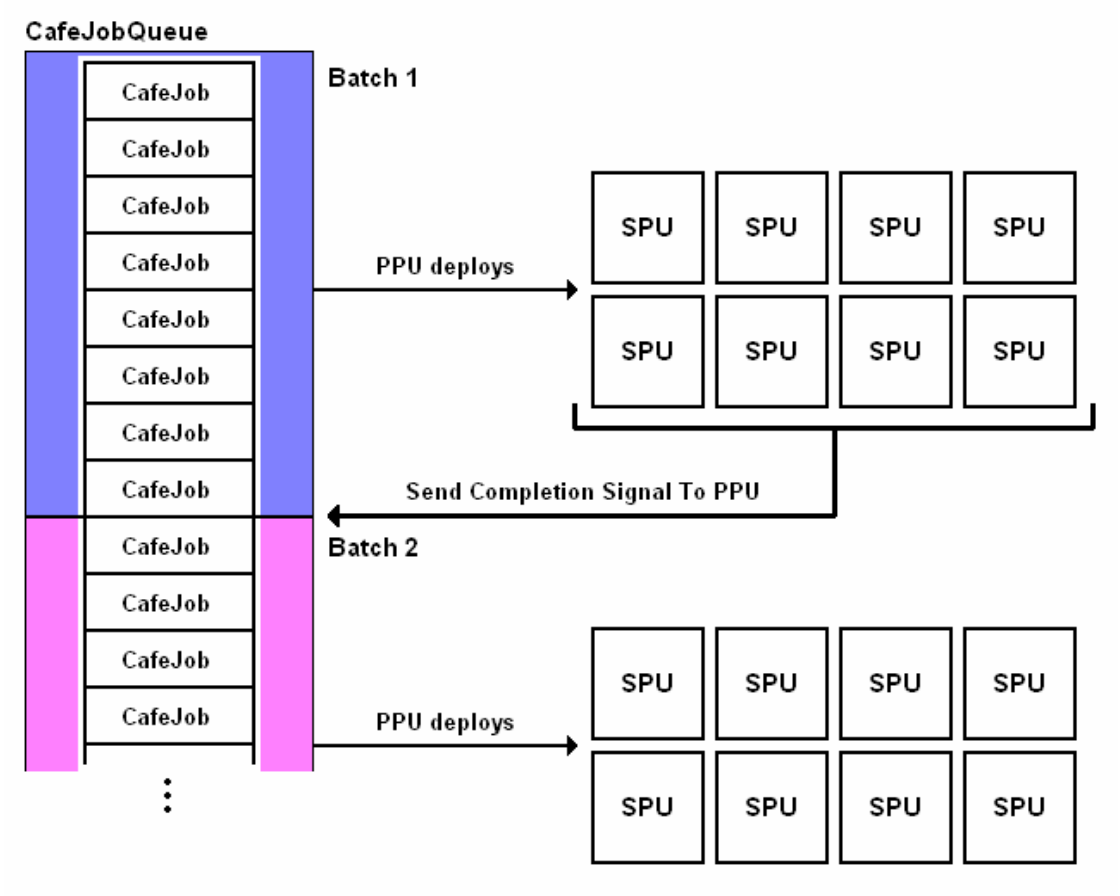



**Figure 5.2: Batch-Sync Programming Model Pipeline**

Moreover, it is important to make the following observation: between the time we dispatch a batch of `CafeJob`s to the SPUs and the time we receive their completion signal, the PPU is free to perform other work. Consequently, we have allowed for the client to split the scheduler into two distinct components: one to deploy a batch of `CafeJob`s and the other to wait on their completion. This way the client is free to use the PPU while the SPUs are processing.

To clarify, when we say "wait on a SPU completion signal" we do not mean in terms of the `spe_wait()` function which actually waits on the return status from the SPU `main()`. In our case, we are simply polling the out mailbox of each SPU for a completion message. Once the mailbox is filled, we can read it and ensure our `CafeJob` was completed successfully, otherwise we should abort or handle the error somehow (this is dependent upon the client and application). Then, once all of the current jobs have reported in successfully we can start up a new batch.

Meanwhile, on the other side of the pipeline each SPU waits until it receives an event message via mailbox to proceed or terminate (we refer to these events as `CONTINUE` and `QUIT`, respectively). If signaled to proceed, the SPU retrieves a control block for a single job payload, carries out the necessary processing for that payload, and returns to the state of waiting on the PPU for its next task.

**Batch-Sync Template**

To better clarify the underlying details, we have provided pseudocode for both the PPU and SPU sides of the Batch-Sync method on the next page. Furthermore, we note that we will discuss a concrete example of this programming model in the first application presented in Chapter 6 in hopes to help the reader draw parallels between the conceptual paradigm and an actual implementation.

Also, in this template we have assumed that the SPU only has code for a single task. However, this model can easily be converted to apply to a SPU program that encapsulates multiple tasks by changing the `CONTINUE` message to be a task-specific continue message (i.e. `CONTINUE_TASK1`, etc.). CAFE does not force the developer to

use any particular scheme in this area of application design, so they are free to structure this message pump in any way that best suits the needs of the application.

**PPU Steps:**

1) Wrap dataset with `MemoryRegion`
2) Create `CafeJobQueue` with `MemoryRegion` and the max `CafeJob` size allowed
3) Create SPU threads
4) Loop over jobs:
    a. Prepare batch of jobs, where the number of jobs in a single batch is found by the following formula:
    $$batchSize = \min(spuThreadCount, remaining\ \ jobs\ in\ queue)$$
    b. For each job in the batch:
        i. Send `CONTINUE` message (via mailbox)
        ii. Send `CafeJob` address (via mailbox)
    c. Do whatever on PPU while jobs are running
    d. For each job in the batch:
        i. Poll the SPU out mailbox until signaled it has been filled
        ii. Read out mailbox and check completion signal
5) Loop over SPU threads:
    a. Send `END` message (via mailbox)
    b. Wait on thread final status

**SPU Steps:**

1) Loop while message is `CONTINUE`
    a. Read in `CafeJob` address
    b. Retrieve `CafeJob` payload (via DMA) from PPU
    c. Perform data processing
    d. Send back processing results (via DMA) to PPU
    e. Send `CafeJob` completion signal
2) Terminate SPU thread

To summarize, first we must generate the `CafeJobQueue` from the dataset. Steps 1 and 2 accomplish this by first wrapping the dataset in a `MemoryRegion` and then (as depicted in Figure 5.1) partitions the data into SPU-sized chunks via the `CafeJobSplitter`. The output of the splitter is the desired `CafeJobQueue`. Next, we spawn the SPU threads (if they have not already been created) in Step 3.

Step 4 encompasses the heart of the Batch-Sync model's DMA pipeline. Here we loop over the `CafeJob`s in the `CafeJobQueue` and dispatch the aforementioned batches, whose size is computed by the formula given in Step 4a. By "dispatch" we actually only need to send the SPU a pair of messages:

1) a `CONTINUE` event message notifying the SPU that it has a task to process,
2) the address to the `CafeJob` so that the SPU can retrieve the input buffer.

It should be noted that while our outline demonstrates the use of only a single `CafeJobQueue` (and therefore uses only one input buffer), it is possible to use multiple queues to provide multiple input buffers for the same task. For example, if we had multiple input streams as in the scenario exemplified in Figure 4.7, it would be convenient to create three separate `CafeJobQueues`, one for each component buffer, and then the address for one `CafeJob` from each queue would have to be dispatched at this time (Step 4b).

At this point (Step 4c), the PPU is free to perform any other tasks as it waits for this batch of `CafeJob`s to complete.

Finally, we need to collect the completion signals from the SPU mailboxes which tell us that the `CafeJob`s have been processed and the SPUs have gone back to waiting for the next event signal (Step 4d).

When the jobs in the `CafeJobQueue` have been exhausted, we can clean up the SPU threads. This is accomplished by sending a `QUIT` event message to the SPUs which forces them to break from their idle waiting and return a final status.

On the SPU side, things are a bit simpler. First, we establish the message pump by looping on a `CONTINUE` signal from the PPU. If a `CONTINUE` signal is given, then we must retrieve the address to the `CafeJob` and transfer the data payload. It is here that we use the `CafeJobDmaManager` or employ a custom method of data transfer. Next, we perform the data processing and send back the results (which can also be done via the `CafeJobDmaManager`). The final step is to send a completion signal and go back to waiting for the next event message. When the PPU sends a `QUIT` message, we simply clean up and return the appropriate exit code.

**Batch-Sync Code Sample**

Next we provide code example that follows the pseudocode template above.

**PPU Example Code:**

```
// Givens:
const int kMaxCafeJobSizeInBytes;         // available space on SPU
const int kElementCount;                  // float vector size
const int kSpuCount;                      // number of SPUs to use

//-------------------------------------

// allocate data buffer
const int kBufferSizeInBytes = kElementCount * sizeof(float);
float* dataBuffer = (float*)aligned_malloc(kBufferSizeInBytes, 128);

// fill data buffer here ...

// create CafeJobQueue
MemoryRegion memRegion(dataBuffer, kBufferSizeInBytes);
CafeJobQueue* jobQueue = GenerateJobQueue(memRegion,
      kMaxCafeJobSizeInBytes, sizeof(float));
```

```
// spawn SPU threads
speid_t* speids = new speid_t[kSpuCount];
for(int spuIdx = 0; spuIdx < kSpuCount; ++spuIdx)
   speids[spuIdx] = spe_create_thread(0, &SpuProgram, 0, 0, -1, 0);


// Batch-Sync pipeline
const int kJobCount = jobQueue>GetJobCount();
for(int jobIdx = 0; jobIdx < kJobCount; )
{
   // compute batch job count
   const int batchJobCount = Min(kSpuCount, kJobCount-jobIdx);


   // send jobs in batch
   for(int batchJobIdx = 0; batchJobIdx < batchJobCount;
++batchJobIdx)
   {
      // send continue
      spe_write_in_mbox(speids[batchJobIdx], (unsigned int)CONTINUE);

      // send CafeJob address
      spe_write_in_mbox(speids[batchJobIdx],
          (unsigned int)&jobQueue->mJobs[jobIdx]);

      ++jobIdx;
   }


   // PPU free to do work here ...

   // wait for jobs in batch to finish
   for(int batchJobIdx = 0; batchJobIdx < batchJobCount;
++batchJobIdx)
   {
      while(!spe_stat_out_mbox(speids[batchJobIdx]));
      spe_read_out_mbox(speids[batchJobIdx]);
   }
}


// tell SPUs to finish
for(int spuIdx = 0; spuIdx < kSpuCount; ++spuIdx)
{
    spe_write_in_mbox(speids[spuIdx], (unsigned int)QUIT);

    int status;
    spe_wait(speids[spuIdx], &status, 0);
}


// cleanup
delete [] speids;
aligned_free(dataBuffer);
```

**SPU Example Code:**

```
Givens:
const int kDmaTag;
const int kJobCompletionTag;
byte dataBuffer[MAXBUF];      // where MAXBUF is some max buffer size

//-------------------------------------

CafeJob cafeJob;

while(spu_read_in_mbox() == CONTINUE)
{
   // retrieve CafeJob metadata
   unsigned int ppuCafeJobAddr = spu_read_in_mbox();

   // retrieve CafeJob
   CafeJobDmaHelpers::DmaCafeJobFromPpu(cafeJob, ppuCafeJobAddr,
kDmaTag);

   // retrieve data buffer from PPU
   const int dataBufferSizeInBytes = cafeJob->GetElementCount() *
sizeof(float);
   MemoryRegion memRegion(dataBuffer, dataBufferSizeInBytes);
   CafeJobDmaHelpers::InitiateDmaFromPpu(memRegion, cafeJob,
kDmaTag);
   CafeJobDmaHelpers::FinalizeDmaFromPpu(kDmaTag);


   // data processing here ...


   // send results back to PPU
   CafeJobDmaHelpers::InitiateDmaToPpu(memRegion, cafeJob, kDmaTag);
   CafeJobDmaHelpers::FinalizeDmaToPpu(kDmaTag);

   // notify PPU we finished this CafeJob
   spu_write_out_mbox(kJobCompletionTag);
}
```

On the PPU, we first allocate and fill our data buffer. In this case, we are creating a dataset of `float`s, aligned to a 128-byte boundary. Certainly these lines could be swapped out for loading the dataset from disk, etc. After the dataset has been prepared, we must wrap it in a `MemoryRegion` so that the CAFE data-partitioning routines can split the dataset into SPU-sized payloads via the `CafeJobQueueGenerator::Generate()` routine. Now we have a `CafeJobQueue`.

Next, we must spawn the SPU threads and begin processing the `CafeJob`s of the `CafeJobQueue` in batches (which we have labeled as the "Batch-Sync pipeline"). First, we compute the size of the batch by finding the minimum of the number of SPU threads and the remaining jobs (as per the formula given in the template). Each job is dispatched via a pair of mailbox messages; we note that the address of the job is readily accessible via the `CafeJobQueue` member `mJobs`.

When the SPU threads have finished, they will send out a completion message to the PPU that must be read. We have done this by looping over the jobs from the batch and continually polling the out-mailbox for each SPU; `spe_stat_out_mbox()` will return true when the out-bound mailbox has been filled, allowing us to read the completion message. While we have not done so here, the completion message may be checked to ensure the SPU processing completed successfully—the details to this check remain application-specific. Once the entire batch has completed we repeat the pipeline steps until all of the `CafeJob`s in the queue have been processed.

The final PPU step is to clean up the SPU threads and data buffer memory. Since the SPUs are still waiting for a message from the message pump, we must tell them to quit and then wait on their return code via the `spe_wait()` function.

On the SPU, the message pump is simulated by a while loop whose condition is to check whether the next message is to continue. This corresponds to the first mailbox message sent in the batch loop. Inside this loop, we must first obtain the address to the `CafeJob`, so that we may DMA the `CafeJob` to the SPU via the `CafeJobDmaHelpers::DmaCafeJobFromPpu()` routine. This routine fills the `CafeJob` structure passed in using the given `kDmaTag`.

Now that we have a `CafeJob` we can retrieve the payload it represents via the `CafeJobDmaHelpers::InitiateDmaFromPpu()` routine, which makes all of the DMA requests for all of the chunks of data that make up the payload and places them in the data buffer wrapped by the given `MemoryRegion`. We follow this with a call to `CafeJobDmaHelpers::FinalizeDmaFromPpu()`, which executes the wait on the DMA tag used during the requests; it should be noted that double-buffering may be employed by interleaving the data processing between the initiate and finalize routines. Once the finalize routine returns, the entire payload has been transferred to the SPU.

Once the data has been processed, we can transfer the payload back to the PPU in a manner similar to which it came. We note that we are assuming that the same payload is stores the results, though this is not a requirement; if a different buffer that was not previously transferred from the PPU is used to store the results, then a separate `MemoryRegion` and `CafeJob` must be created.

The final step is to send the completion message, signaling the PPU that we have finished processing this `CafeJob` payload. Again, this message may be customized for error-handling purposes.

Before concluding this subsection, we would like to point out that by following the Batch-Sync model we are actually still imposing a level of serialization by moving down the `CafeJobQueue` one batch at a time. This could be improved further by immediately sending out a new job once a SPU has completed its current one, but this has the disadvantage that now the PPU must be burdened with responding to these signals continually instead of at given increments. Our solution to this problem is

presented in the next section on the second programming paradigm which we call Subqueue Streaming.

### 5.3.2 Subqueue Streaming Programming Model

The *Subqueue Streaming* programming model eliminates the aforementioned level of serialization that is imposed by stepping through the job queue at the rate of a single batch at a time. Subqueue Streaming (or sometimes just "Streaming") pushes the responsibility of job retrieval onto the SPUs as opposed to leaving it on the shoulders of the PPU as in the Batch-Sync approach.

However, without a coherent manager to oversee the distribution of tasks, an important question emerges: how do the SPUs know which jobs in the job queue are theirs to process? The key is to realize that if we partition the entire `CafeJobQueue` into $k$ contiguous subqueues, then all we have to do is send off the first `CafeJob` in each subqueue to a corresponding SPU and inform it of the number of jobs it is responsible for in its designated subqueue. This works because the SPU can infer where to find each next job in its subqueue since all of the `CafeJob`s in the original queue were created in a contiguous block of memory, allowing us to find each next job through the use of some pointer arithmetic.

In fact, we found that this operation of partitioning `CafeJobQueue`s into a given number of subqueues is a very useful pattern to employ in general. Therefore, our framework includes a helper routine to partition any `CafeJobQueue` into $k$ subqueues according to a block cyclic distribution. In the case that the number of `CafeJob`s in the original queue does not divide evenly into $k$ subqueues, we simply

stack the earlier subqueues higher. For example, if our original job queue is of length 18 and we wanted to partition it amongst $k = 8$ SPUs, then each subqueue would be assigned 2 jobs with the exception of the initial two, which would be assigned 3 jobs each. Therefore our subqueue sizes would be: $\{3 \quad 3 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2 \quad 2\}$.

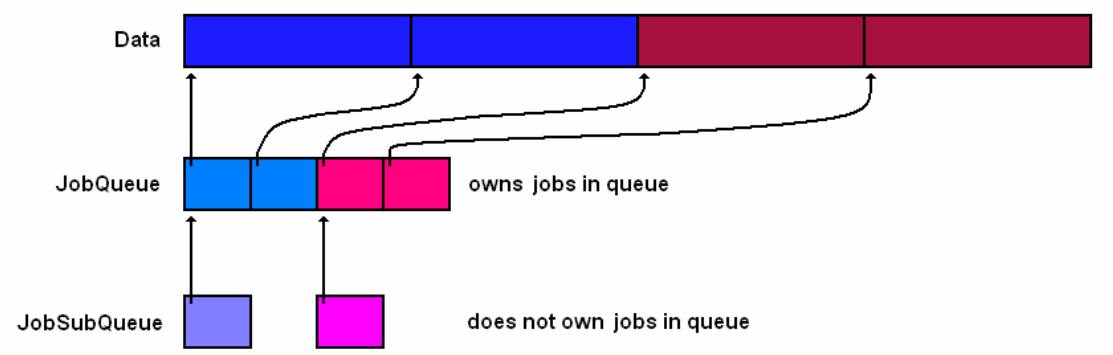



**Figure 5.3: `CafeJobQueue` Partitioning into Subqueues**

It is important to note that when using the `CafeJobQueue` partitioning routine, the subqueues returned (also of type `CafeJobQueue`) do *not* have ownership of the jobs they point to, as indicated by the diagram in Figure 5.3. Instead, they merely point to a contiguous sequence of `CafeJob`s that were originally allocated by the initial `CafeJobQueue`. The idea behind this is that the initial `CafeJobQueue` was created from a buffer that may need to be partitioned into different SPU groupings; i.e.: the first partitioning may be for all 8 SPUs while the next time it may only need to be partitioned for 4. This was one of many solutions, but at the end of the day provided the client with the most flexibility. The only caveat is that the developer must take care not to delete the original job queue as it will also free all the jobs it contains, thus nullifying the subqueues pointing to them.

To be absolutely clear, the primary benefit to not owning the `CafeJob`s for the subqueues is that the original `CafeJobQueue` can be reused for subsequent tasks. Since it is possible for different tasks to exploit different levels of parallelism within an application, we believed it to be important to allow developers to have this freedom to group tasks differently via subqueues.

**Subqueue Streaming Template**

We now present a pseudocode template for the Subqueue Streaming method. Also, a concrete implementation of this paradigm will be discussed alongside the Batch-Sync model in the following chapter.

**`PPU Steps:`**

1) Wrap dataset with `MemoryRegion`
2) Create `CafeJobQueue` with `MemoryRegion` and the max `CafeJob` size allowed
3) Create `CafeJobQueue` subqueues with the subqueue splitter routine
4) Create SPU threads
5) Loop over SPU threads:
    a. Send `CafeJob` address (via mailbox) of first job of designated subqueue
    b. Send count of jobs in designated subqueue
6) Do whatever on PPU while jobs are running
7) Loop over SPU threads:
    a. Wait on thread final status

**SPU Steps:**

1) Read in `CafeJob` address
2) Read in job count
3) Loop over jobs of subqueue:
    a. Retrieve `CafeJob` payload (via DMA) from PPU
    b. Perform data processing
    c. Send back processing results (via DMA) to PPU
    d. Compute next job offset
4) Terminate SPU thread

As in the Batch-Sync model, the first task is to generate the `CafeJobQueue` from the dataset; as such, Steps 1 and 2 are identical to the Batch-Sync template. Next, we must establish the subqueues for each SPU from this `CafeJobQueue` (Step 3). Step 4 spawns the SPU threads (if they have not already been created).

Steps 5 through 7 encompass the core of the Subqueue Streaming model. Notice that these steps are simplified from the corresponding steps in the Batch-Sync template. This is most certainly because the PPU is no longer the administrator of `CafeJob`s to the SPUs. Step 5 dispatches the subqueues to the SPUs by sending the initial `CafeJob`s in the subqueues and the subqueue size. At this point, the PPU is free to perform any other work in parallel with the SPU tasks. Finally, we must collect the return codes from the SPU threads when the subqueue tasks have completed.

One of the main differences on the SPU side is that we no longer have to setup a message pump to receive event signals. The message pump was necessary because the SPU was not aware of which jobs from the `CafeJobQueue` it should process.

Instead, it was fed jobs on an as needed basis. As a result, the Streaming Subqueue SPU template looks quite different from the Batch-Sync version.

First, the SPU must retrieve the subqueue data dispatched by the PPU (as outlined in Steps 1 and 2). Afterwards, we have enough information to loop over and process the `CafeJob`s in this SPU's designated subqueue. The inside of this loop looks similar to the Batch-Sync version with the exception of the final step which is to compute the offset of the next `CafeJob` in the subqueue rather than signal the PPU that the job has been processed. Then, finally after all of the `CafeJob`s in the subqueue have been processed we can clean up and return the thread exit code.

**Subqueue Streaming Code Sample**

Next, we provide code example using the CAFE API that follows the pseudocode template above.

**PPU Example Code:**

```
// Givens:
const int kMaxCafeJobSizeInBytes;         // available space on SPU
const int kElementCount;                  // float vector size
const int kSpuCount;                      // number of SPUs to use

//-------------------------------------

// allocate data buffer
const int kBufferSizeInBytes = kElementCount * sizeof(float);
float* dataBuffer = (float*)aligned_malloc(kBufferSizeInBytes, 128);

// fill data buffer here ...

// create CafeJobQueue
MemoryRegion memRegion(dataBuffer, kBufferSizeInBytes);
CafeJobQueue* jobQueue = GenerateJobQueue(memRegion,
   kMaxCafeJobSizeInBytes, sizeof(float));
```

```
// create subqueues
CafeJobQueue* jobSubqueues = new CafeJobQueue[kSpuCount];
jobQueue->PartitionIntoSubqueues(jobSubqueues, kSpuCount);


// spawn SPU threads
speid_t* speids = new speid_t[kSpuCount];
for(int spuIdx = 0; spuIdx < kSpuCount; ++spuIdx)
   speids[spuIdx] = spe_create_thread(0, &SpuProgram, 0, 0, -1, 0);


// Subqueue Streaming pipeline
for(int spuIdx = 0; spuIdx < kSpuCount; ++spuIdx)
{
   // send subqueue info

   // subqueue job count
   spe_write_in_mbox(speids[batchJobIdx],
      jobSubqueues[spuIdx].GetJobCount());

   // address of first job in subqueue
   spe_write_in_mbox(speids[batchJobIdx],
      (unsigned int)&jobQueue->mJobs[jobIdx]);
}


// PPU free to do work here ...


// tell SPUs to finish
for(int spuIdx = 0; spuIdx < kSpuCount; ++spuIdx)
{
   int status;
   spe_wait(speids[spuIdx], &status, 0);
}


// cleanup
delete [] speids;
delete [] jobSubqueues;
aligned_free(dataBuffer);
```

**SPU Example Code:**

```
Givens:
const int kDmaTag;
byte dataBuffer[MAXBUF];      // where MAXBUF is some max buffer size

//-------------------------------------

CafeJob cafeJob;

// retrieve subqueue metadata
const unsigned int kJobCount = spu_read_in_mbox();
const unsigned int ppuCafeJobAddr = spu_read_in_mbox();


int offsetInBytes = 0;
for(int jobIdx = 0; jobIdx < kJobCount; ++jobIdx)
{
   // retrieve CafeJob
   CafeJobDmaHelpers::DmaCafeJobFromPpu(cafeJob,
      ppuCafeJobAddr + offsetInBytes, kDmaTag);


   // retrieve data buffer from PPU
   const int dataBufferSizeInBytes = cafeJob->GetElementCount() *
sizeof(float);
   MemoryRegion memRegion(dataBuffer, dataBufferSizeInBytes);
   CafeJobDmaHelpers::InitiateDmaFromPpu(memRegion, cafeJob,
kDmaTag);
   CafeJobDmaHelpers::FinalizeDmaFromPpu(kDmaTag);


   // data processing here ...


   // send results back to PPU
   CafeJobDmaHelpers::InitiateDmaToPpu(memRegion, cafeJob, kDmaTag);
   CafeJobDmaHelpers::FinalizeDmaToPpu(kDmaTag);

   offsetInBytes += sizeof(cafe::WarpJob);
}
```

The code for the Subqueue Streaming implementation proceeds by creating `CafeJobQueue` from data buffer wrapped by a `MemoryRegion`. Next, we must create a list of `CafeJobQueue`s (in the code: `jobSubqueues`) to serve as the storage for the subqueues. This list is passed to the `PartitionIntoSubqueues()` routine along with the number of SPU threads (which is the number of subqueues we want to generate).

Next, we spawn the SPU threads and begin the Subqueue Streaming pipeline. For each SPU, we send out the number of `CafeJob`s in the corresponding subqueue in addition to the address to the first `CafeJob` in the subqueue. There is no particular ordering to this data, just so long as it matches with the receiving calls in the SPU code. At this point, the SPUs are cranking on all of the jobs in the subqueue so the PPU is free to perform other tasks in the meantime.

At last, we must check the return codes from each of the SPU threads. Recall that the SPU programs in the Subqueue Streaming model run to completion once dispatched with a subqueue, so there is no need to wait on a completion message here. Finally, we must clean up the memory used (be sure to include the subqueue list!).

On the SPU, we wait for the PPU to send the subqueue metadata: the number of `CafeJob`s this SPU is responsible for processing and the address to the first job in our subqueue. While looping over the jobs in our subqueue, we must keep an offset from the address of the first `CafeJob`, so that we can proceed through the queue without explicitly asking the PPU for this information. We use this offset when we call the `CafeJobDmaHelpers::DmaCafeJobFromPpu()` routine to retrieve the `CafeJob` from the PPU and fill the `cafeJob` structure.

With this `CafeJob`, we can now retrieve the data payload via the initiate and finalize DMA routines. As before, double-buffering may be employed by interleaving computation between these calls. Finally, when we have finished processing we must send our results back (again, assumed to be stored in the same buffer as the input data). After the results have been sent, we increment our job offset so we can retrieve the next `CafeJob` in the queue.

As one might suspect, the SPU-driven job management approach featured in the Subqueue Streaming model outperforms the PPU-supervised scheme utilized by the Batch-Sync pipeline (see Section 6.1.3). Despite this difference, we have still found important uses for the Batch-Sync programming model, namely when the jobs are not guaranteed to be processed in a uniform manner since we have the opportunity to disrupt and dynamically adjust the state of the overall task according to the results received. This is not possible with the Streaming method—once the SPUs have been dispatched with their subqueues they are committed to processing those jobs.

## 5.4 Summary

In this chapter, we introduced the Cell Architecture Framework and Extensions (CAFE) library: a minimalistic framework designed to assist developers in taking advantage of the computational power provided by the Cell's SPUs without forcing a single programming model onto their applications. CAFE is focused on providing a direct solution to data partitioning and data transfer, yet still allows developers to keep explicit control over the scheduling decisions of the flow of data.

First, we explained our initial set of design goals for CAFE: we wanted a lightweight, flexible, and non-intrusive framework that keeps the developer close to the hardware. In practice, we were able to accomplish this by employing a metadata type to serve as the link between the application code and our framework library, in addition to designing an open data transfer pipeline which allows developers to swap out stages within the data partitioning and data transfer process. As a result, we believe our framework to take a unique approach.

Next, we discussed the structures and utilities provided by the CAFE API to help developers partition their data into SPU-sized payloads and dispatch them to the SPUs. In CAFE, the core abstraction is called a *job*, which represents a unit of work over a SPU-sized payload of data. A `CafeJob` does not actually own this data to avoid the performance and memory overhead incurred by a deep copy. A `CafeJobQueue` can automatically be created from a dataset using one of the generators provided in the framework. CAFE also supplies the appropriate utilities for transferring these jobs to and from the SPUs in both a single- and multi-buffered fashion. Under the hood, we have employed the data transfer technique described in our discourse on DMA in Chapter 4 in an effort to maintain the best performance possible.

Thereafter, we outlined a pair of programming models which utilize the mechanisms provided in CAFE: the Batch-Sync programming model and the Subqueue Streaming programming model. In the Batch-Sync paradigm, the PPU dispatches batches of `CafeJob`s to the SPUs from a `CafeJobQueue` created from an input dataset. Meanwhile, the SPU sits idle waiting for this job in a message pump; the SPU will either be notified that there is a job waiting or that it should terminate. If a `CafeJob` is dispatched, the SPU will retrieve the corresponding data payload, perform the appropriate processing, send back any results necessary, and return to waiting in the message pump. While slower than the Subqueue Streaming model, the Batch-Sync paradigm allows the PPU to employ checks and balances to alter the flow of data to the SPUs if needed. In the Subqueue Streaming model, the `CafeJobQueue` created from the input dataset is further partitioned into subqueues which are dispatched to each of the SPUs. In this paradigm, the SPUs are responsible for

retrieving the jobs themselves, as opposed to waiting on the PPU to coordinate the distribution of jobs; in other words, there is no need for a message pump because the tasks have already been divided amongst the processor prior to their transmission. Effectively, this means that once the subqueues are dispatched, the jobs will run to completion without the opportunity for interruption or reorganization.

**Chapter 5 Notes**

1. It should be noted that the `CafeJobSplitter` has no way of knowing how much actual memory is available on the SPU, so it will generate jobs for any arbitrary size it is given.
2. The size of a SPU binary can be determined with the `spu-size` program in the `spu-binutils` package. Usage: `spu-size <binary_file>`.
3. All CAFE mechanisms assume and utilize 128-byte alignment when partitioning data and creating `CafeJob`s.
4. The stride is one of the primary indications for why we state that our framework is geared towards stream-processing: we expect input buffers to uniformly consist of the same structure types; otherwise a general partitioning scheme would be much more difficult to implement and use in practice.

## Chapter 6: Example Applications and Results

In the previous chapter, we presented a pair of base templates for developing an application using CAFE. Now, in this chapter we will outline actual applications created with our framework which employ these templates and discuss the results.

It is our intention to be thorough in the implementation details in an effort to give the reader better insight into the utilities provided by the framework. While each of the following applications was implemented and run on a Cell BladeServer, we note that only a single Cell processor was used to gather the results. Since CAFE places no restrictions on the number of SPUs used during program execution, any of the following applications can easily be extended to utilize both processors on a blade.

### 6.1 SAXPY

The primary pair of application implementations we will discuss stem from a very simple operation: the Scalar Alpha X Plus Y, or more commonly referred to as SAXPY. SAXPY calculates the following expression: $z[i] = y[i] + \alpha \cdot x[i]$, for all $i$ in the associated arrays $x$, $y$, and $z$.

We have chosen this particular application for a few reasons. First, it is a standard benchmark computation across the High-Performance and Scientific Computing communities, and thus familiar. It is also a very common and useful operation that is usually a component part to a much larger algorithm. Finally, the individual computations are all independent of each other, allowing us to exploit the operation in a data-parallel manner.

In terms of implementation, we present both Batch-Sync and Subqueue Streaming versions of SAXPY. Throughout, we will discuss the applications in terms of the framework and present the appropriate pseudocode. Finally, we will compare the runtime results to confirm the assertions from Section 5.3.2.

### 6.1.1 Batch-Sync SAXPY

The Batch-Sync implementation is fairly straightforward. It follows the pipeline model described in Section 5.3.1 as shown in the pseudocode in Figure 6.1. We also note that the code itself follows the sample source provided in Chapter 5 almost verbatim, with the exception being the fact that here we now have two input buffers and thus a pair of corresponding `CafeJobQueue`s.

```
PPU:
 Given datasets x and y, both of same length n.
 Create CafeJobQueues for both x and y.
 Start k SPU threads.
 Loop while there are still jobs left in the queue:
    Send out metadata for a batch of jobs.
    Wait for all jobs in batch to return.
 Terminate SPU threads.

SPU:
 Loop while PPU has more jobs in queue:
     Retrieve job metadata.
     Initiate DMA requests for both x and y buffers.
     Issue Tag Sync.   // DMA is complete here
     Data Processing: z[i] = y[i] + alpha*x[i].
     Send back result buffer.
```

**Figure 6.1: Batch-Sync SAXPY Pseudocode**

When we create the `CafeJobQueue`s for the $x$ and $y$ vectors, we must inform our data-splitter of the stride to use and the maximum size for the payload that each job can contain. In this case, the stride is simply the size of the data type in each vector (which could be `int`, `float`, `double`, or any object with the appropriate operator overloads). Since we must fit three buffers of equal size on the SPU to perform this operation (one for $x$, $y$, and the result buffer $z$), we can compute the maximum job size according to: $\left(256KB - \left(size\ of\ SPU\ code\right)\right) / 3$.

In the job queue loop on the PPU, the metadata sent out includes the `CONTINUE` event message and an address to a control structure. We note that this differs slightly from the outline in Chapter 5 since we have more information to deliver to the SPU than just a single `CafeJob` address. The control structure holds the job addresses as well as the count of the elements to be processed during this SPU job. Granted, we could send these off in a sequence of mailbox messages, but we find it much more convenient (in terms of both implementation and maintenance) to DMA a control structure rather than managing a set of mailbox transfers that must also adhere to the limits of the mailbox queue depth.

Although the Batch-Sync method is a nice, simple model that will certainly exploit data-parallelism in an intuitive manner, it has one critical flaw as we pointed out in previous chapter: the SPUs must synchronize with the PPU so that the batches are all processed in lock-step. This serialization is an unnecessary constraint for our particular application. As such, we will now examine a Subqueue Streaming implementation of SAXPY.

### 6.1.2 Subqueue Streaming SAXPY

Our SAXPY implementation can be accelerated if we reorganize how we send off our jobs into a more stream-like manner. In other words, instead of the PPU continually designating which job each SPU is to process next, we can precompute this sequence of jobs. As we saw in Section 5.3.2, the most convenient way to partition the job queue is to designate a contiguous sequence of jobs for each SPU so that each subsequent job address can be computed by adding an offset to the address of the initial job (as opposed to explicitly transferring the address as we do in the Batch-Sync method). As outlined in the pseudocode below, we make use of CAFE's subqueue partitioning routine to automatically handle this preprocessing task.

```
PPU:
 Given datasets x and y, both of same length n.
 Create CafeJobQueues for both x and y.
 Partition both CafeJobQueues into subqueues.
 Start k SPU threads.
 For each SPU thread:
    Send out metadata for job subqueues.
 Wait for all jobs in batch to return.
 Terminate SPU threads.

SPU:
 Retrieve job subqueue metadata.
 Loop while more jobs in subqueue:
     Initiate DMA requests for both x and y buffers.
     Issue Tag Sync.    // DMA is complete here
     Data Processing: z[i] = y[i] + alpha*x[i].
     Send back result buffer.
```

**Figure 6.2: Subqueue Streaming SAXPY Pseudocode**

We would like to draw our reader's attention to the differences between the loops on the PPU in the two implementations. First, the Batch-Sync method must not only dispatch the metadata control structure for each job (handled by an inner loop over the processors associated with the current batch of jobs), but also must wait for each SPU to signal its completion before continuing (handled in a separate secondary inner loop). However, in the Subqueue Streaming method we are able to eliminate the outer loop and produce a much cleaner (and faster!) result.

This central change echoes in SPU implementation. Since the Subqueue Streaming method is no longer dependent upon the supervision of the PPU, the main loop changes from an event-driven loop to a loop over the jobs in the subqueue, which also requires the control structure retrieval to be performed outside the loop.

### 6.1.3 SAXPY Results

While it stands to reason that the Subqueue Streaming method is faster than the Batch-Sync method by intuition, we now provide the appropriate analysis to answer the question of how much faster it can be.

To start, we present Figure 6.3 below that delineates the total runtime for the Batch-Sync and Subqueue Streaming SAXPY SPU implementations. Note that the *x*-axis represents the size of each of the vectors in terms of elements where $K = 1024$ and $M = (1024 \times 1024) = 1048576$.

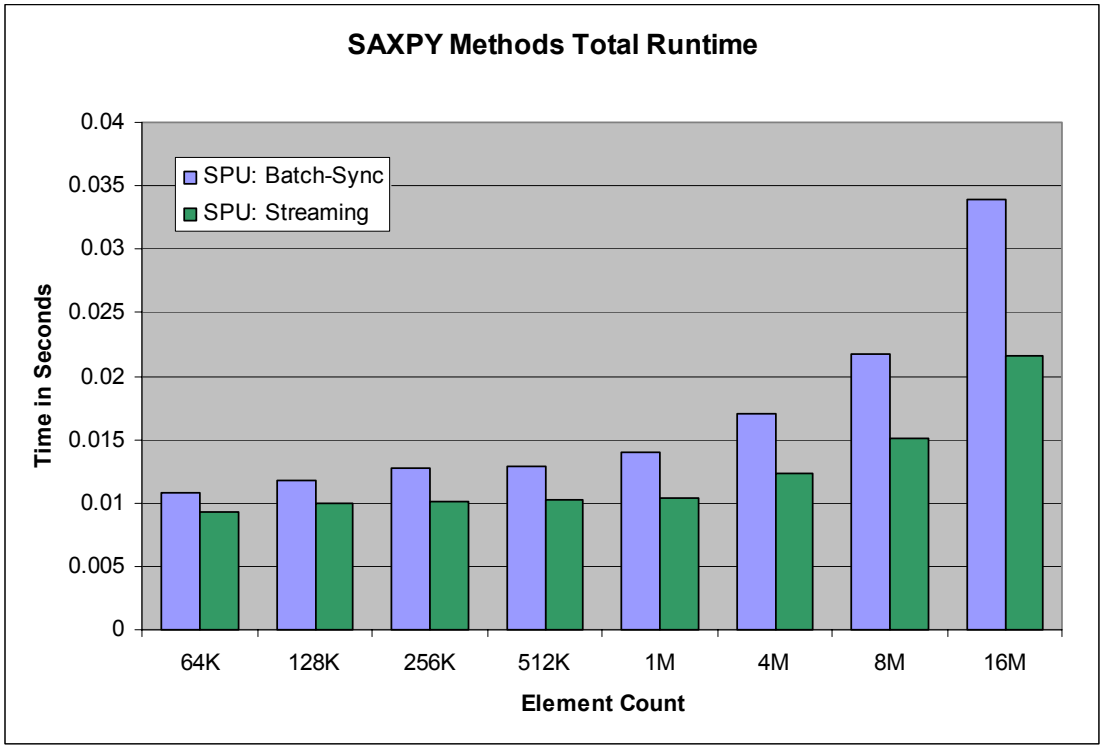



**Figure 6.3: SAXPY SPU Implementation Runtimes**

The runtimes shown above should come at no surprise. We can see that before a payload size of a million elements (in this case `floats`) the SPU implementations hover around the same runtime. This is most certainly because of the SPU thread spawning (for 8 SPUs) and job queue initialization times that are added on top of the data processing. Furthermore, the affects of this additional setup time is echoed in the relative performance between the SPU implementations and the PPU implementation, shown in Figure 6.4. However, after a million elements we begin to observe significant speedup.

Moreover, we have also included a plot of the relative speedup between the two SPU implementations in Figure 6.5 to confirm our intuitions of the supremacy of the Subqueue Streaming method over the Batch-Sync method.

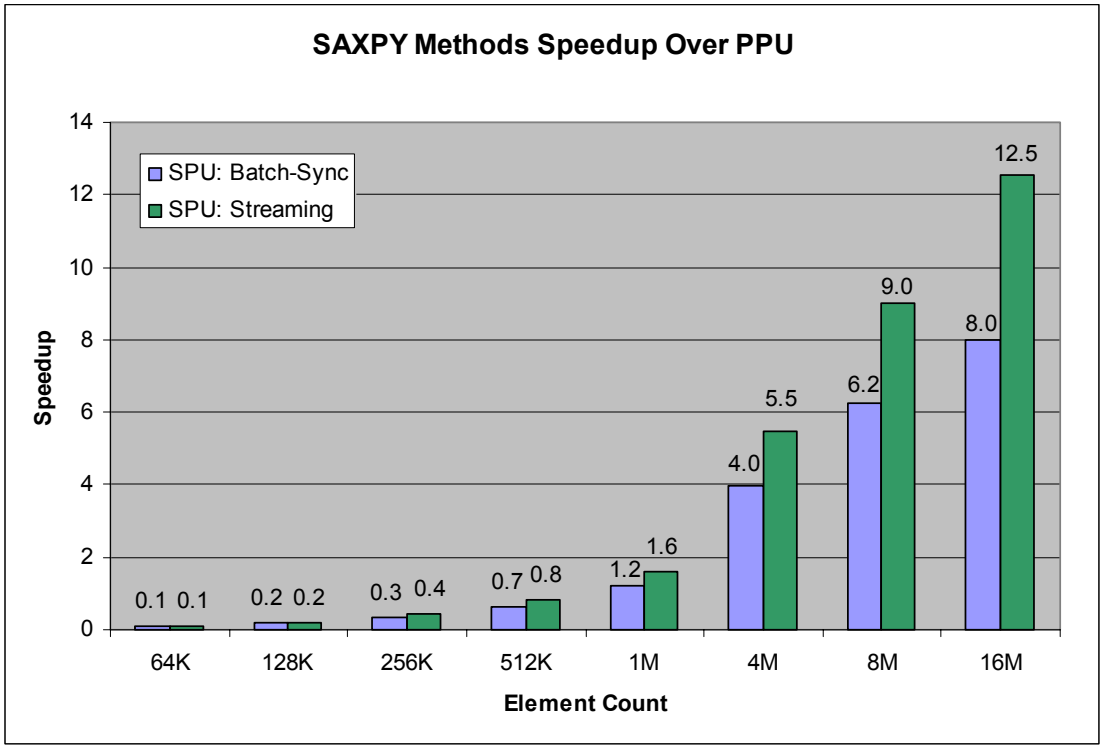



**Figure 6.4: SAXPY SPU Speedup over PPU**

First, we would like to draw the reader's attention to the 12.5*x* speedup of the SPU Subqueue Streaming method over the PPU in the final benchmark of Figure 6.4 as a demonstration of the possibility for *super-linear speedup* with Cell. This is namely due to the difference in data caching between the PPU (standard memory hierarchy) and SPU (no cache, just local store). Effectively, the combined amount of data sitting in the local stores across the SPUs is 1.2 MB (150 KB each) which translates to enough data for 400 K SAXPY operations before it has to be switched out. This is much larger that the 32 KB cache size on the PPU. Thus, this means that the PPU must fetch data approximately five times more often than each of the SPUs since they each obtain their data payloads all at once, which allows them to perform the computation for a longer period without interruption.

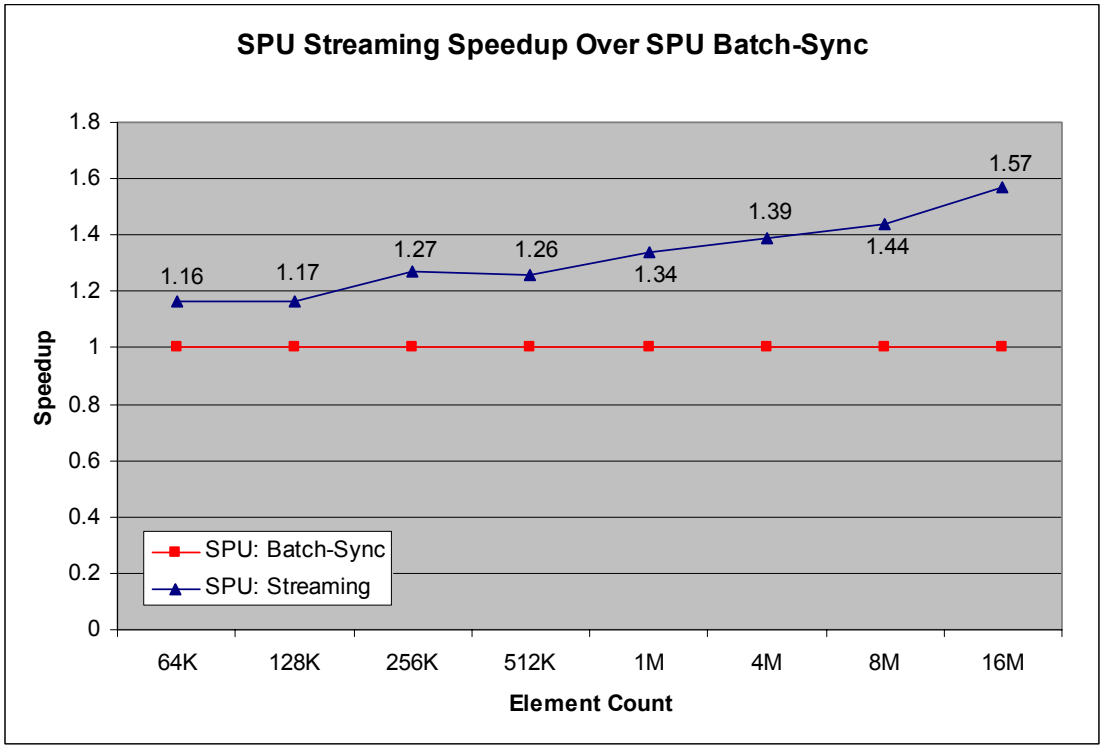



**Figure 6.5: SAXPY SPU Streaming Speedup over SPU Batch-Sync**

Second, we note that in this application we have observed that the Subqueue Streaming method can render up to a 1.5*x* speedup over the Batch-Sync method. Additionally, the upward trend shown in Figure 6.5 is caused by the increasing number of `CafeJob`s generated to process the increasing size of the dataset. In the Batch-Sync method, each additional `CafeJob` comes with an additional overhead of dispatching and waiting for that job to complete. However, the Subqueue Streaming method does not incur this additional overhead since each SPU is responsible for its own set of `CafeJob`s and does not require the supervision the PPU nor does it require the rest of the SPUs to complete their jobs in lock-step. Therefore, we project that this factor of improvement of the Subqueue Streaming method over the Batch-Sync

method will continue to grow in cases of more complicated computational operations where processing times on the SPUs may have a higher variance since it would require the PPU to wait longer to complete the synchronization step before dispatching another batch.

Furthermore, although it is not shown here, we also advocate the employment of multi-buffering to layer the computation over the latency caused by the data transfer. We project that combining this technique with the Subqueue Streaming approach to the implementation would further improve performance since it would allow the SPU to almost be in a continual state of computation.

**Total Time**
these numbers are # of elements

| | 64K | 128K | 256K | 512K | 1M | 4M | 8M | 16M |
|---|---|---|---|---|---|---|---|---|
| PPU | 0.0011 | 0.0021 | 0.0042 | 0.0084 | 0.0169 | 0.0675 | 0.1353 | 0.2708 |
| SPU: Batch-Sync | 0.0108 | 0.0117 | 0.0128 | 0.0129 | 0.0140 | 0.0171 | 0.0217 | 0.0338 |
| SPU: Streaming | 0.0093 | 0.0100 | 0.0101 | 0.0103 | 0.0104 | 0.0123 | 0.0151 | 0.0216 |

**GFLOP RATE**

| | 64K | 128K | 256K | 512K | 1M | 4M | 8M | 16M |
|---|---|---|---|---|---|---|---|---|
| PPU | 0.9977 | 0.9967 | 0.9932 | 0.9930 | 0.9940 | 0.9938 | 0.9921 | 0.9914 |
| SPU: Batch-Sync | 0.0969 | 0.1793 | 0.3277 | 0.6491 | 1.2016 | 3.9309 | 6.1906 | 7.9308 |
| SPU: Streaming | 0.1128 | 0.2092 | 0.4166 | 0.8173 | 1.6061 | 5.4578 | 8.9181 | 12.4339 |

**Figure 6.6: SAXPY Implementation Runtimes and GFLOP Rates**

Finally, we note that we were able to reach 12.4 GFLOPS with the Subqueue Streaming implementation as shown in Figure 6.6. We believe that this is a modest showing for our application since we gave no special attention to tuning in terms of optimal performance, simply because the main goal for this application was to demonstrate the implementation level differences across the programming models using CAFE. In the light of this circumstance, we believe that additional

modifications (such as the aforementioned multi-buffering) could be employed to further improve the performance on top of the current implementations.

We conclude this section with the final line counts for each of the SAXPY implementations in Figure 6.7 below.

| | PPU | SPU | Total |
|---|---|---|---|
| PPU | 131 | 0 | 131 |
| SPU Batch-Sync | 232 | 60 | 292 |
| SPU Streaming | 244 | 69 | 313 |

**Figure 6.7: SAXPY Implementation Line Counts**

## 6.2 Ray Tracer

The next application we present is a simple ray tracer. We chose this application because it requires a significant amount of computational work. It also is comprised of multiple stages, two of which can be parallelized. These characteristics will allow us to demonstrate CAFE's effectiveness and non-intrusive nature in a sizable application.

### 6.2.1 Ray Tracing, Briefly

For those not familiar with the fundamentals of ray tracing, we will briefly present a high-level overview of the algorithm. For more information see [Shirley02].

The brute-force approach to ray tracing can be summed up in the following algorithm (see Figure 6.9 for illustration):

```
Brute-Force Ray Tracing Algorithm
for each pixel
{
   compute primary ray
   for each triangle
   {
      if primary ray intersects triangle
         shade pixel
   }
}
```

**Figure 6.8: Serial Brute-Force Ray Tracing Algorithm**

The application uses two datasets: the image data and the scene data. While the image data is stored as a grid of pixels in the final output image, each pixel initially starts off as a "primary ray" from the eye / camera that passes through a virtual image plane. The scene data is composed of a list of triangles, materials, and lights. Conventionally, the geometry is read in from one or more model files from disk, each of which can contain thousands to millions of triangles. It is then transformed to its desired location and orientation in the scene. Each triangle in a mesh has an associated material to describe its appearance under the influence of the lights within the scene.

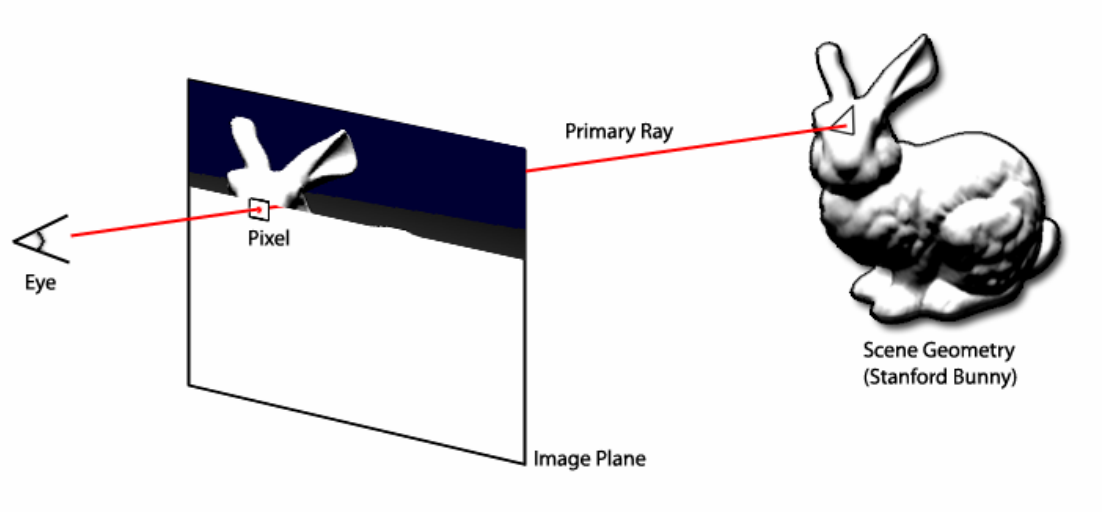



**Figure 6.9: Ray Tracing in Action**

Computing the primary ray requires the knowledge of a viewing position and direction as well as the dimensions of the image plane. The image plane is broken up into a grid structure according to the resolution of the output image in pixels. A ray is constructed to start from the viewer's position and passes through the center of each pixel square.

To test for an intersection of a ray and triangle, the ray is not only checked for an intersection within the bounds of the triangle, but is also checked to be closer than any other previous intersection along the path of that ray, since an object closer to the eye will block the view of any other objects directly behind it. Provided both conditions hold, we say that we have a *valid intersection* and can shade that pixel with the color of the associated triangle's material. If there is an intersection, the result is a `HitInfo` structure that stores the data required by the shader to compute the final output color.

The bulk of the time spent by this algorithm is within the ray-triangle intersection test. Thus, it has been the subject of many research papers as to how to alleviate this bottleneck through the use of acceleration structures. However, in the interest of simplicity no acceleration structures were used in our implementation[1]. This also allows us to deterministically compute the number of ray-triangle intersection tests that must be performed to generate an $n \times m$ image:

*Ray-Triangle Intersection Test Count* $= (n \times m) \times triangleCount$ .

### 6.2.2 Ray Tracing on Cell

In our implementation, the PPU loads the appropriate scene data and manages the two parallel stages: the first consists of the primary ray computations and the ray-triangle intersection testing, while the second shades the corresponding image pixels. Consequently, we have packaged the ray-triangle intersection test and shader into separate SPU binaries. This makes it convenient to swap out the intersection testing method or shader for a different one with minimal code change. Furthermore, when the scene is read in from file a corresponding list of shaders is assembled; we can then use this list to dynamically associate the correct SPU shader binary with each triangle mesh according to requirements of the scene.

It is easy to imagine that we can split the image into a number of slices and since no pixel depends on any other we know that each slice itself is independent. Therefore, we can simply run the same serial brute-force ray tracing algorithm on each slice. As a result, we have setup our ray tracer pipeline (shown in Figure 6.10 below) to split the image into $k$ slices, where $k$ is the number of available SPUs. Each SPU

is initialized with the ray-triangle intersection binary and is sent a metadata control block that contains the camera information and a set of bounds that delineate the slice of the image plane that the SPU is responsible for processing. At this point, the SPUs are equipped to compute the primary rays and perform the ray-triangle intersection tests required for a small subset of the image at a time. This subset is a portion of a `HitInfo` result buffer on the PPU and can be partitioned with the mechanics provided in the CAFE library. Scene geometry is DMAed for intersection testing in accordance with the Subqueue Streaming paradigm introduced in Section 5.3.2 and the resulting `HitInfo` structures are stored into the output buffer and sent back to the PPU as they are completed.

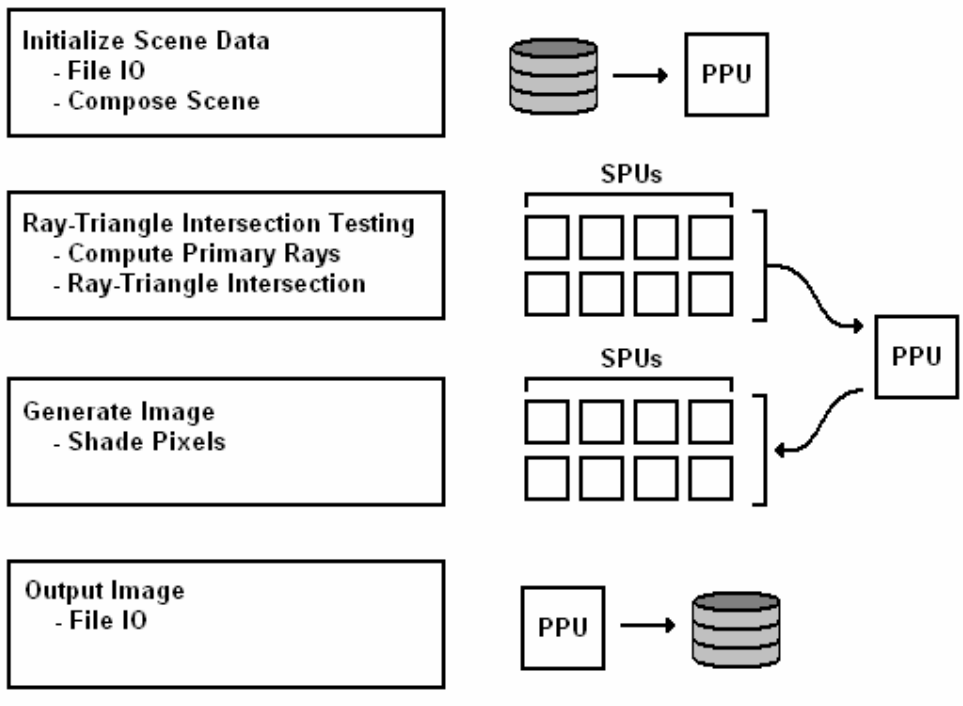



**Figure 6.10: Ray Tracer Pipeline on Cell**

Before continuing on to the next stage of the pipeline, we should highlight a primary difference between a ray-triangle intersection test on a conventional processor like the PPU and the same test on the SPU. Normally, an intersection test on a conventional processor takes advantage of employing early rejection measures to avoid any unnecessary computations, making it a branch-heavy yet more efficient operation. However, the opposite approach is taken in the SPU implementation, which leads to a more computationally-intensive but branchless operation, better catering to the SPU's defining strength. This implies that through our partitioning each SPU is given an identical workload size in both data and computation, each totaling to $\underbrace{(n \times m / k) \times triangleCount}_{Data\ Size} \times \underbrace{80\ vector\ operations}_{Ray-Triangle\ Computation}$.

Once all $k$ SPUs have completed this stage, the PPU terminates the Ray-Triangle Intersection threads and launches a new set of SPU threads with the appropriate shader binaries. At this point, we can now split the buffer of `HitInfo` structures again and loop over the job queue as usual. This repartitioning of the `HitInfo` structures buffer is performed because the shader binaries do not require as much data to be present on the SPU to complete each operation as the ray-triangle intersection since no scene geometry is needed. Thus, only image data is needed and so we can fit more pixels into the SPU local store. For example, in the ray-triangle intersection routine we might only be able to process 20 rows of pixels of the image at a time, while in the shader we might be able to process 40 rows of pixels at a time. This scenario also underscores the versatility of the level of the structures provided by

the CAFE framework as it allows developers to partition their datasets in ways best-suited for each task at different stages within their application.

### 6.2.3 Ray Tracer Results

We now present experimental results comparing the running times of our PPU and SPU ray tracer implementations, as shown in Figure 6.11 and Figure 6.12. We used three test scenes of varying sizes, each rendered at a resolution of $512 \times 512$, or 256 K pixels. It should be noted that we have plot the results of the two implementations separately due to the large differences in magnitude.

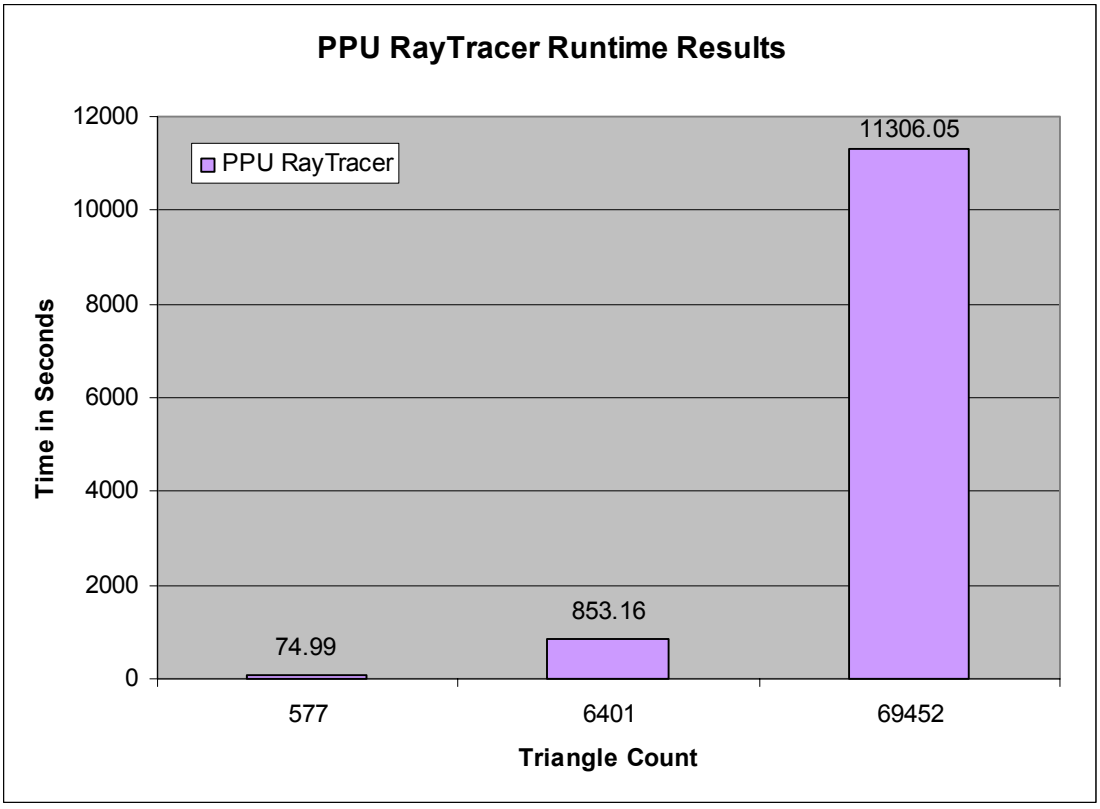



**Figure 6.11: PPU Ray Tracer Runtime**

We implemented both a Batch-Sync and a Streaming implementation for the SPU. Interestingly, the differences in the running times differed by an insignificant amount due to the overwhelming computational cost found in the ray-triangle intersection testing routine. In fact, under further investigation, we found that the ray-triangle intersection testing accounts for almost 90% of the application runtime in our SPU implementations. As such, we project that the employment of acceleration structures would bring forth differences between the methods similar to those we observed in the SAXPY implementation comparison. This is because acceleration structures cut down on the amount of computation needed per pixel, potentially giving the communication component a larger percentage of the application runtime.

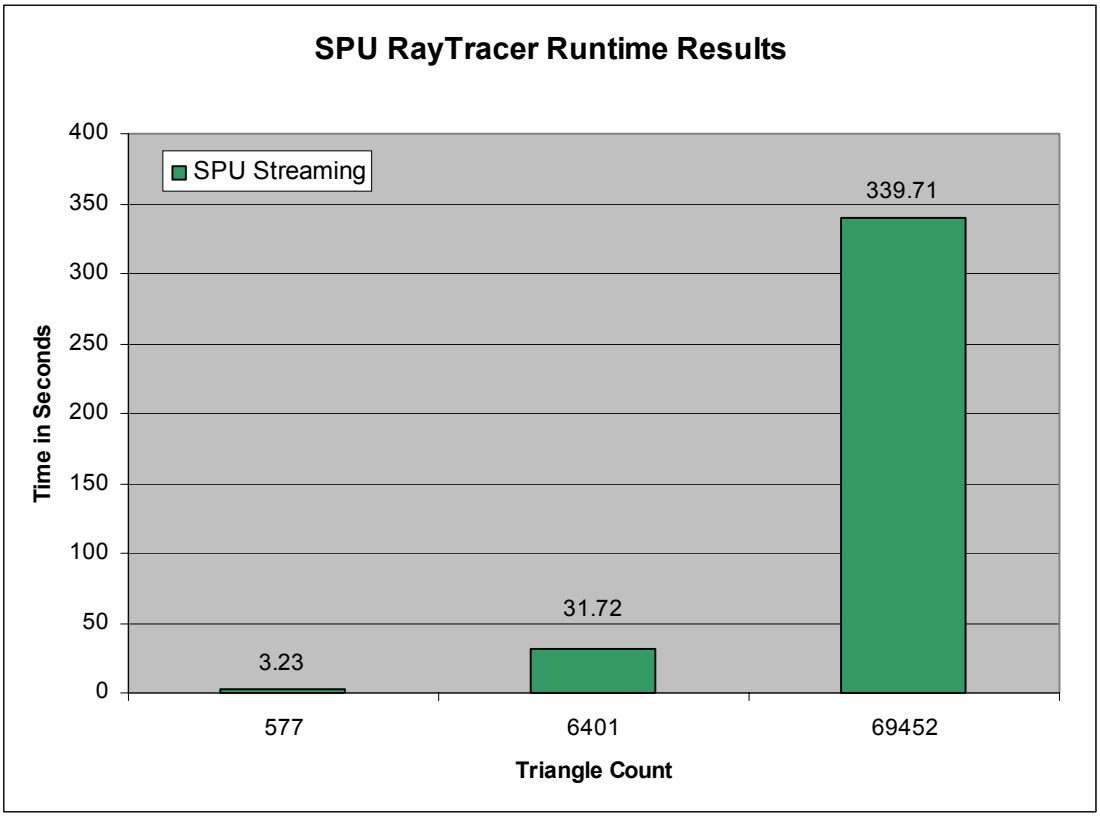



**Figure 6.12: SPU Ray Tracer Runtime**

The SPU ray tracer measurements shown above boast a range from 23-33% performance improvement over the PPU implementation. Similar to our previous application, we note that the reason behind the super-linear speedup stems in part from the difference between the size of combined local storage across all eight SPUs (again, 150 KB each = 1.2 MB total) and the size of PPU's cache (32 KB) as explained in Section 6.1.3. Additionally, this improvement also stems from the increased speed of the SPU's single-precision floating-point operations over the PPU.

To better see these differences in runtime performance, we present the relative speedup of the SPU Subqueue Streaming implementation over the PPU implementation in Figure 6.13.

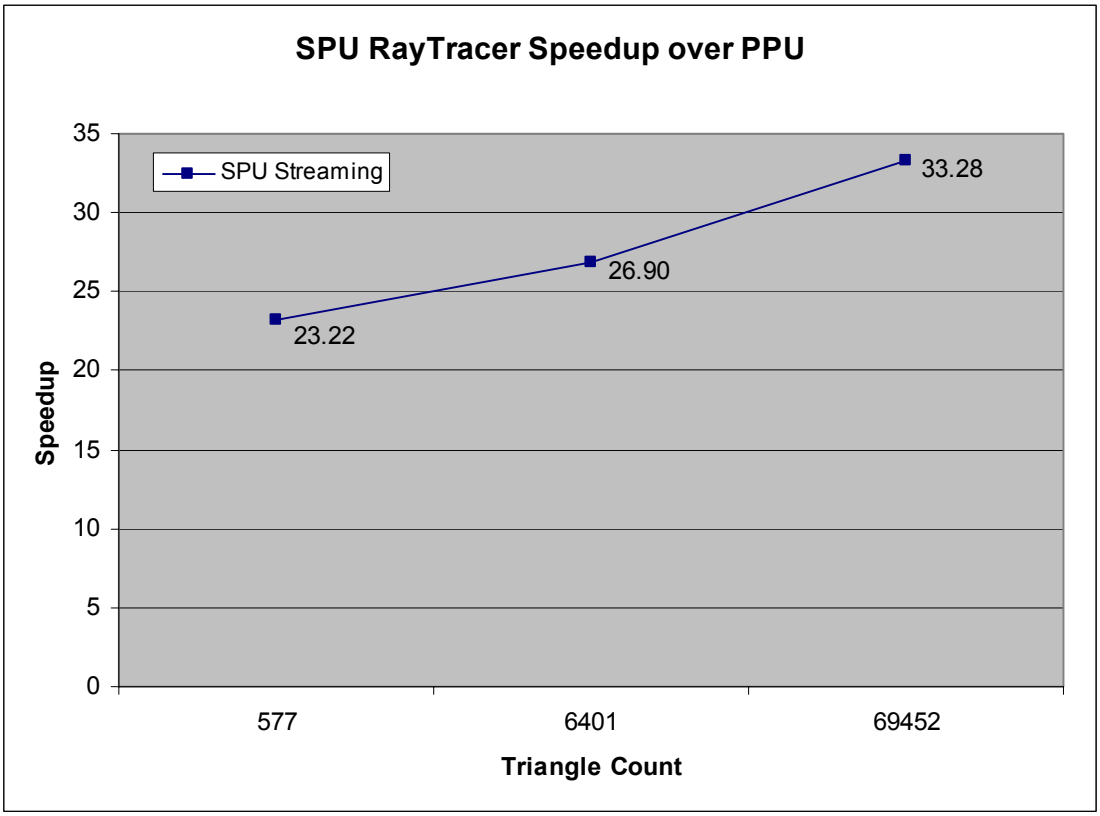



**Figure 6.13: SPU Ray Tracer Speedup over PPU**

We note, however, in order to gain this performance improvement via the use of the SPUs an additional 740-790 lines of code were required. This is not only due to the additional `CafeJobQueue` setup and processing, but also to the different nature of the algorithms for the ray-triangle intersection and shader on the SPU. The final line counts for our implementations are given in Figure 6.14 below.

| | PPU | SPU Ray-Tri | SPU Shader | Total |
|---|---|---|---|---|
| PPU | 3540 | 0 | 0 | 3540 |
| SPU - Batch-Sync | 3900 | 211 | 212 | 4323 |
| SPU - Streaming | 3855 | 214 | 215 | 4284 |

**Figure 6.14: Ray Tracer Implementation Line Counts**

## 6.3 Image Filtering via Chaining

One programming model we have continued to promote throughout this work is Chaining. We believe that this is an important and misrepresented programming paradigm for Cell that certain applications can map onto in both a clean and intuitive manner. Therefore, we would like to investigate this method in an application to not only show its usefulness but also to demonstrate CAFE's ability to support more than just the Function-Offload programming model.

To demonstrate, we consider the application of a pair of filters to a given image to produce a subsequent output image. For the sake of simplicity, we will only discuss a chain of two SPUs, though we could extend the same principles to a chain utilizing any number of SPUs supported by the system.

### 6.3.1 Parallel Image Processing

In this section we present the basic concept and process behind applying a filter to an image. We also discuss the necessary requirements for parallelization. However, readers are encouraged to see [Gonzalez02] for a more thorough discourse on the topics of image processing.

In image processing, the process of applying a filter to an image is carried out by performing the same local computation upon each pixel of an input image to produce a corresponding output image. An image filter (also commonly referred to as a *kernel*) is a square block of values conventionally of odd dimensions (this way there is a distinct center pixel corresponding to a resultant pixel in the output image). This grid of values is convolved with the pixels of the input image to produce the pixel values of the final output image. The convolution operation works in the following manner: the values in the filter are used as weights for the corresponding pixel values in the image and these products are all summed into a single result value. A simple example of this procedure with a $3 \times 3$ kernel is shown in Figure 6.15 below.

There are many types of filters that may be applied to an image, including those used for blurring, sharpening, edge-detection, color level adjustment, etc., and in most cases this same process of "sliding" a kernel over an image is employed in one form or another. Also, kernels can vary in size as well as in depth (of color channels); however, for the sake of simplicity we shall limit our discussion to $3 \times 3$ kernels which only operate on a single color channel at a time, though the work that follows can most certainly be extended to the needs of more complicated filters.

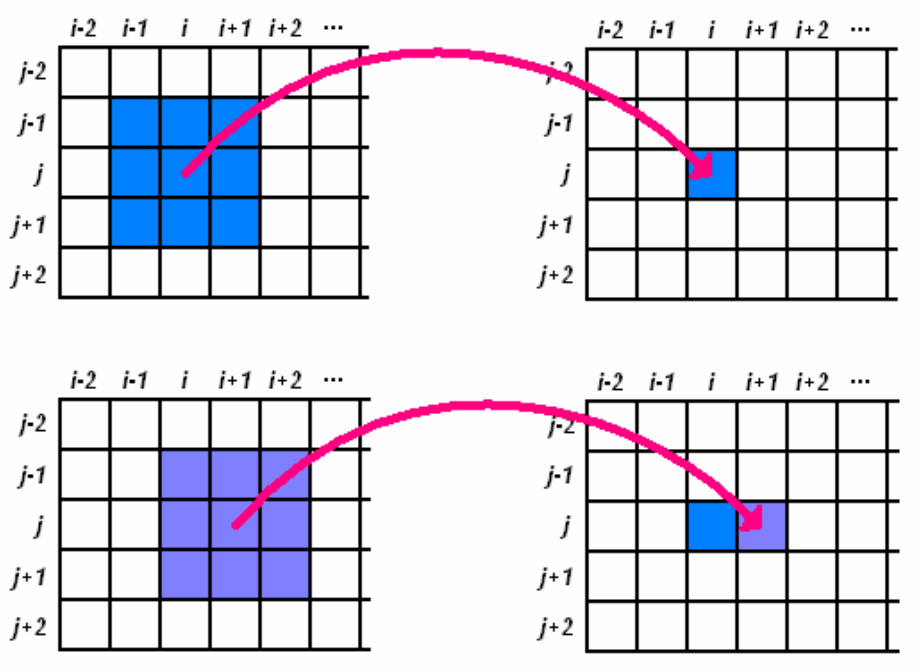



**Figure 6.15: Sliding a 3x3 Kernel across the Pixels of an Image**

It is important to realize that this is actually a very different type of operation than the ones we examined in the previous sections. In particular, this routine requires the values of the neighboring elements of the particular element under consideration. In other words, although each element can be processed individually (i.e. it is still a data-parallel operation), the set of data elements this operation is dependent upon spans a small window within the dataset (as opposed to just a single element). To put this in terms of our $3 \times 3$ image filter, each row of pixels in the image requires the values of the rows above and below it in order to carry out its computation. We also assume in our case that the borders of the image are padded with zero-valued pixels (this is just one of many ways to handle border pixels, see [Gonzalez02] for a complete list of options).

This behavior has repercussions for a parallel implementation. Primarily, when we partition the image data to be spread across the processors, we must in fact include these extra rows to correctly process the image. These *ghost cells* exist only to support the computation of other data and are not considered elements to be processed, only referenced. As such, the data distributed amongst the processors must have overlapping segments in order to properly process the entire image (see Figure 6.16 for a simple example between two processors). From hereon we will refer to this overlapping scheme as *windowed partitioning*.

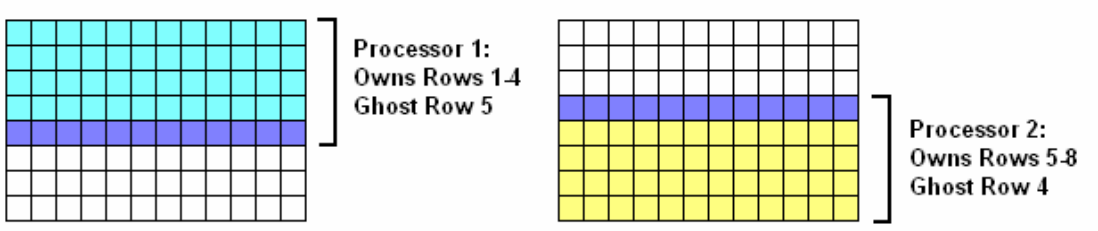



**Figure 6.16: Windowed Partitioning of an Image between Two Processors**

As a matter of fact, image processing is not the only application type that requires ghost cells and the employment of windowed partitioning for an accurate parallel implementation; other applications include stencil methods (i.e. fluid dynamics simulations, heat equation solvers, etc.) and connected component labeling. Therefore, we decided that including an alternative partitioning scheme for creating jobs in this manner would be a valuable asset to provide in the framework.

### 6.3.2 CAFE Support for Windowed Partitioning

The job partitioning scheme we have utilized in each of our previous applications will not suffice for the requirements of the image filtering process. Our current partitioning scheme assumes that each element is not dependent upon any other data within the same dataset, and therefore does not take care to include any extra data than the elements to be directly operated upon during that job.

For this very reason, CAFE provides a secondary data splitter which allows the client to specify the number of ghost rows required to be included in each job. Like the uniform data splitter, it performs the appropriate partitioning within the dataset and produces a `CafeJobQueue`. As such, the job creation pipeline for a windowed partitioning application is no different from the uniform partitioning pipeline previously introduced and exemplified. Each job in the resultant job queue starts and ends at the appropriate places in the dataset to compensate for the ghost cells and holds an offset to the start of the actual data to be processed.

We note that it is also common to perform a windowed partitioning along the columns in addition to the rows—separating the dataset into a grid of sub-blocks. However, unfortunately we cannot support this method of partitioning since `CafeJob`s require their data to be completely contiguous within the dataset because they do not make a copy of the data they encapsulate. Consequently, this restriction implies that the size of a row and its required ghost cells cannot be larger than the SPU local store, putting a limit on the pixel resolution of the image that may be processed with this functionality. As an example, for an image of floating-point colors in RGBA format, each pixel is represented by 16 bytes; we can fit three rows (for a $3 \times 3$ kernel) of

5120 (5K) pixels comfortably in the SPU store: $5120 \times (3 \times 16 \text{ bytes}) = 240 \text{ KB}$. However, we note that we will actually need a secondary buffer (for reasons discussed next), and therefore are actually limited to images of half this width.

### 6.3.3 Chaining Implementation Details

While the essence of the Chaining model lies in the scheduling, there are a few particular components that remain unresolved. We will discuss these details here and outline the basis for a simple Chaining application on Cell.

First, when data needs to be transferred from one SPU to another we must take care not to overwrite the data currently in the subsequent stage in the chain. Thus, the SPUs need to communicate to each other when they have completed their respective workloads. This could be handled in a few different ways, but we decided to achieve this effect via a barrier supervised by the PPU. We have done this for two reasons: first, a barrier can easily be implemented via the familiar mechanism of mailboxes; second, this gives the PPU the opportunity to perform a necessary cleanup operation.

This cleanup operation is required because of the existence of the ghost cells embedded within the payload data. As a result of not making a copy of the data into the `CafeJobs`, the original windowed partitioning data cannot be compromised during the processing since some parts of the data are required in multiple jobs. The upshot is that we cannot directly write to this original buffer at the end of the chain. Therefore, a secondary scratch buffer is needed to store the output. Also, the ghost cells may possibly offset the real data in such a way that it no longer starts on a DMA-compliant byte boundary, meaning that we cannot produce the output image by simply DMAing

only the processed data back to the PPU when we have finished the SPU chain. Our solution was to DMA the entirety of the payload back to the aforementioned scratch buffer on the PPU which can then assemble the processed output data accordingly, avoiding the need to place heavier countermeasures in the job DMA pipeline. We have found this technique to be quite suitable for our image filtering application.

To clarify, we now present a template for a Chaining application using CAFE in Figure 6.17 that employs the techniques outlined above.

| PPU | SPU 1 | SPU 2 |
|---|---|---|
| Partition Image into CafeJobs<br>Start the SPU threads<br>Send first job to first stage in chain | | |
| Loop over jobs: | Retrieve job metadata<br>Loop over jobs:<br>DMA image data from PPU<br>Apply Image Filter | Retrieve job metadata<br>Loop over jobs: |
| Wait at Barrier A | Wait at Barrier A | Wait at Barrier A |
| Resolve output data | DMA payload to second SPU | |
| Wait at Barrier B | Wait at Barrier B | Wait at Barrier B |
| | | Apply Image Filter<br>DMA payload to PPU |
| Terminate SPU threads | | |

**Figure 6.17: Template for Chaining Application using CAFE**

### 6.3.4 Image Filter Results

We now present results from our Image Filter application which follows the template in Figure 6.16. The application applies a pair of blur filters to different channels of the given image and writes the result to file. However, we note that the time to write the final image to file is not included in our measurements below.

First, we measured the running time for applying these two filters to images of varying sizes as shown in Figure 6.18 below. The *x*-axis labels represent both the image width and height at the same time (i.e. 512 represents a $512 \times 512$ image). However, it should be noted that although we only have provided measurements for square images whose dimensions are powers of two, this is not an implicit requirement of the application; images may be of any dimensions in width and height (within the size constraint of the SPU local store as previously discussed).

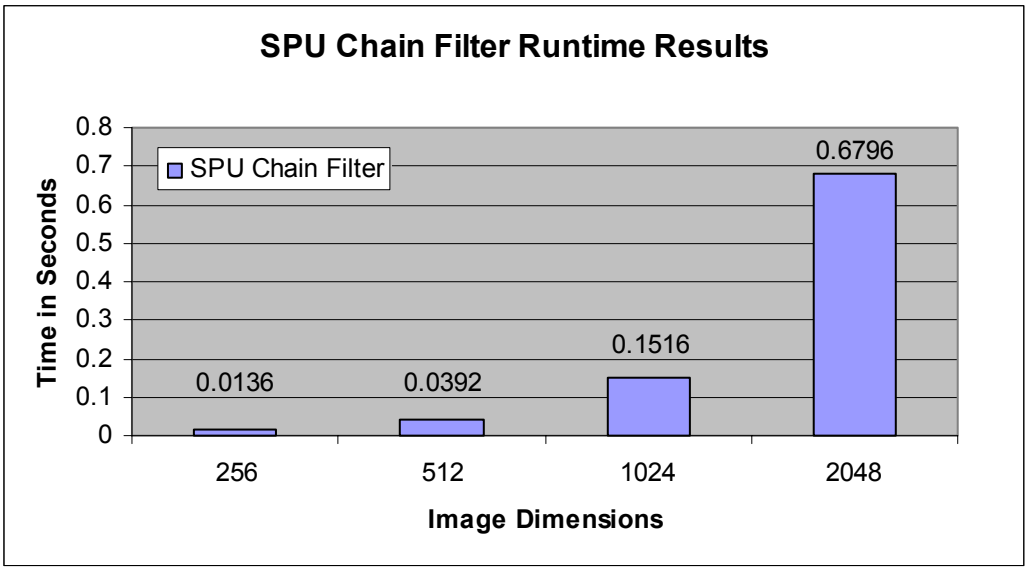



**Figure 6.18: SPU Chain Filter Runtime Results**

Next, we provide a plot of the speedup of the SPU implementation of the image filter over the PPU implementation in Figure 6.19.

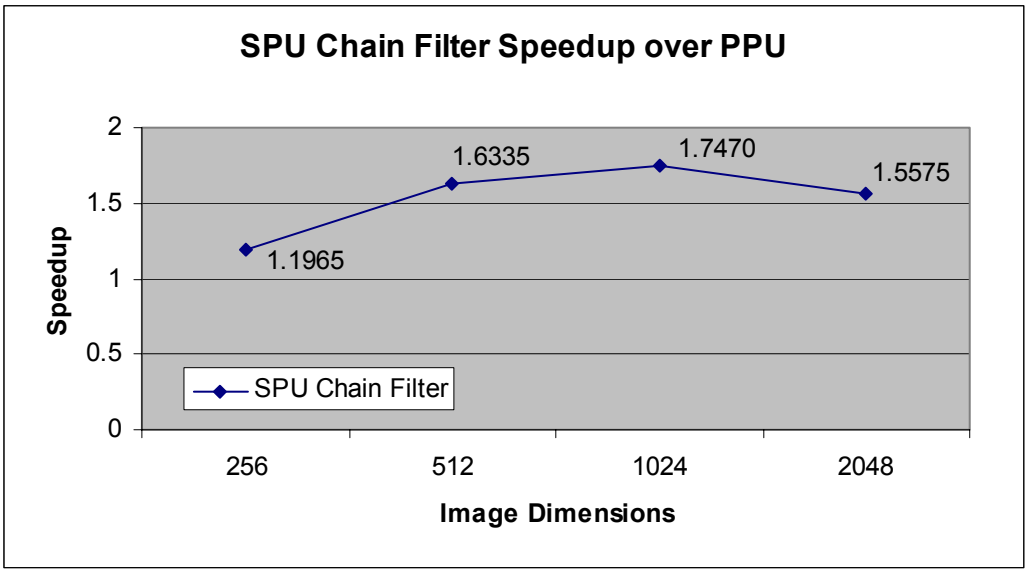



**Figure 6.19: SPU Chain Filter Speedup over PPU (Normalized to 1)**

Here we note that we see a drop in speedup for our last measurement. We believe this to be a direct result of the number of jobs generated for the image (which in fact turns out to be 1 per row). Figure 6.20 shows the size (in bytes) of a row in the image and the corresponding number of `CafeJob`s generated to encapsulate this row in addition to its required ghost rows (one or two rows depending on its position within the image).

| | Row Size | Job Count |
|---|---|---|
| **64** | 1K | 1 |
| **128** | 2K | 3 |
| **256** | 4K | 12 |
| **512** | 8K | 52 |
| **1024** | 16K | 256 |
| **2048** | 32K | 2047 |

**Figure 6.20: SPU Image Filter Statistics**

Finally, we present the relative speedup of the SPU Chain Filter over an Offload implementation in Figure 6.21 below to confirm our previous remarks concerning the benefit of employing Chaining over the Function-Offload programming model for particular applications such as this.

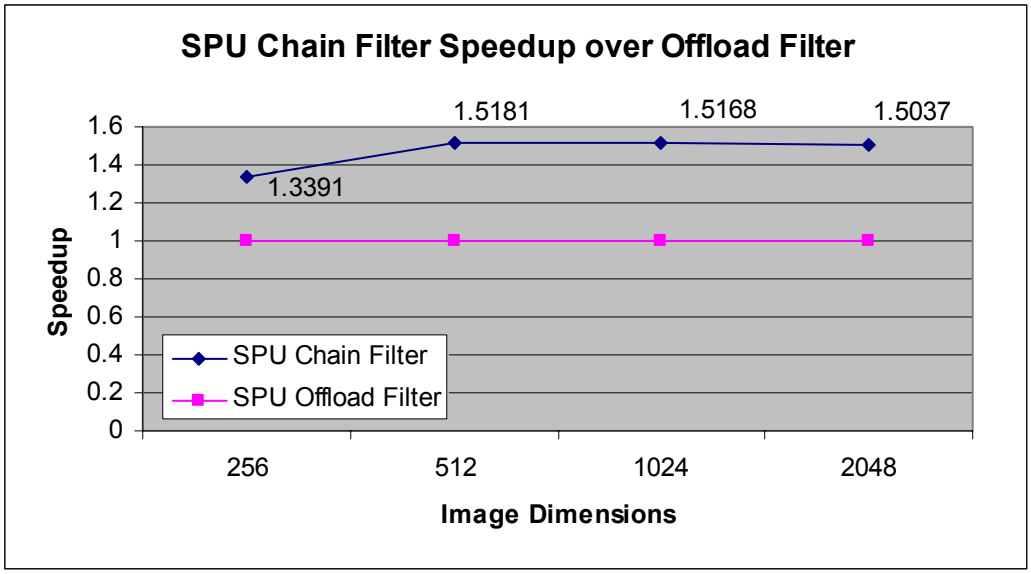



**Figure 6.21: SPU Chain Filter Speedup over Offload Filter**

Here we observe that additional data transfers to the PPU and back inherent to the Function-Offload paradigm can be expensive in comparison to the Chaining model which invokes a more direct route by ferrying the data between the SPUs. Most certainly, Chaining requires developers to add an layer of synchronization on top the SPUs (maintained by the PPU) to ensure that important data is not overwritten; however, it is apparent that it is well worth the effort to gain the improvements in performance such as those depicted above.

## 6.4 Bitonic Sort

The final application we will examine is an example of a port from straight CellSDK code to CAFE. With this example we wish to demonstrate the important principle that while CAFE presents a higher level view of the architectural mechanics exposed via the CellSDK, it does not impair the fundamental low-level control developers have over the system. Furthermore, we would also like to show that there is no performance penalty for using this higher level interface.

As such, we have ported a sorting application originally written by developers at IBM. It is driven by the Bitonic Sort algorithm (also known as a Batcher Sort), which was designed specifically for use in the parallel processing domain. For more details on the Bitonic Sort, we refer our reader to [Sedgewick98], [Christopher06], or their favorite algorithms text.

### 6.4.1 Source Modifications

First and foremost, we would like to clarify that no algorithmic improvements were employed to the code during the port. In fact, the only high-level change made to the implementation was to the method by which the data is transferred between the PPU and SPUs. As a result, the overall change in source is not as significant in volume as it is in refinement. Primarily, low-level commands to the MFC using raw pointers have been replaced with the appropriate equivalent in terms of `CafeJob`s, `CafeJobQueue`s, and `MemoryRegion`s. In other words, after porting the code now reads at a higher level, closer to the abstractions of the actual application and is therefore much easier to grasp, maintain, and debug.

We observe that the original source code dedicates over 50 lines to implementing its own data transfer functions similar to those already available in the CAFE library. This same pair of functions is practically a staple of Cell application development. Most certainly, such code repetition is a strong indication of the need for abstraction and repackaging, which in our case meant into the form of a framework library. We also note that this same pair of functions was the source of many of our early struggles in Cell development. Thus, it was of great value to us to centralize this fundamental pair of operations so that they could not only be easily maintained and tested, but also accessed and utilized by all of our past, present, and future applications.

| | PPU | SPU |
|---|---|---|
| IBM | 692 | 1814 |
| CAFE | 696 | 1783 |

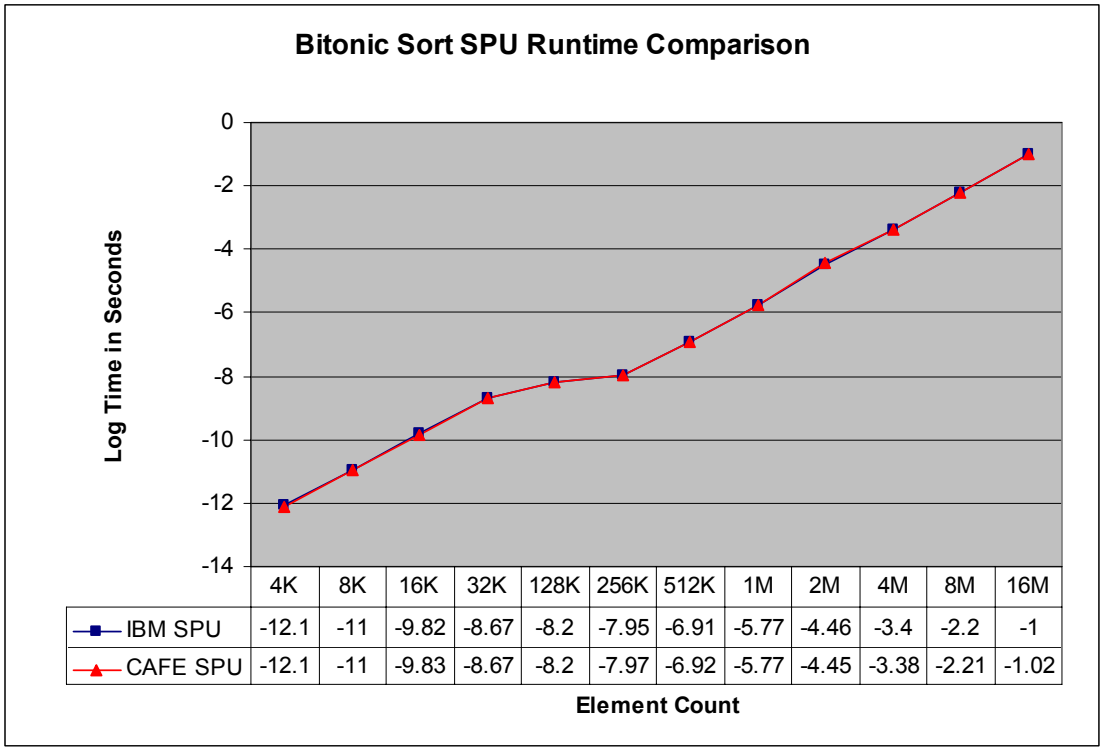
**Figure 6.22: Bitonic Sort Implementations Line Count Comparison**

In Figure 6.22 above, we report the final line count for both implementations. It stands to reason that the extra setup for assembling the `CafeJob`s and their `CafeJobQueue` would require additional code on the PPU side. Also, we see a drop in lines of code corresponding to the exchange of the aforementioned data transfer functions for CAFE's higher level job abstraction. Certainly, data payload retrieval using CAFE's job abstraction does not come free in terms of this measurement; however, in spite of this fact, the overall result is still more compact than the original IBM implementation.

### 6.4.2 Runtime Performance

As one may suspect, the performance differences between the IBM and CAFE implementations are negligible. This is because in both cases the same MFC functions are being called under the hood, allowing us to maintain the same performance available even as if we were using the lowest level of abstraction.

To demonstrate this point, we present the following set of runtime results in Figure 6.23 below, comparing the two SPU implementations for a series of data vectors ranging from 4096 to 16 million elements. Both implementations make use of all eight SPUs to sort a given number of signed integer elements.




| | 4K | 8K | 16K | 32K | 128K | 256K | 512K | 1M | 2M | 4M | 8M | 16M |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IBM SPU | -12.1 | -11 | -9.82 | -8.67 | -8.2 | -7.95 | -6.91 | -5.77 | -4.46 | -3.4 | -2.2 | -1 |
| CAFE SPU | -12.1 | -11 | -9.83 | -8.67 | -8.2 | -7.97 | -6.92 | -5.77 | -4.45 | -3.38 | -2.21 | -1.02 |

**Figure 6.23: Bitonic Sort SPU Runtime Comparison**

From these results above, it is readily apparent that the differences in running time between the implementations are indistinguishable since the measurements continue to overlap throughout. Also, we note that the arc formed up to 256 K elements (equivalent to 1024 KB) is due to the fact that we can still house all of the elements across the combined local store of the SPUs. However, when we make the jump to 512 K elements (2048 KB) we can no longer house all of this data in the SPU local stores (in addition to the code!), which forces the implementation to invoke an out-of-core version of the Bitonic Sort to execute over this dataset.

### 6.4.3 Final Outlook: CellSDK and CAFE

In conclusion, we have demonstrated through this example that we can use the higher level interface provided by CAFE to implement our application in a more intuitive manner, while still achieving performance results similar to those attained by the low-level MFC calls exposed by the CellSDK. We believe that this is one of the most important characteristics of CAFE since it preserves the same level of control and available performance granted by the hardware.

## 6.5 Summary

In this chapter we have examined a few example application implementations utilizing the CAFE framework. Throughout, we have taken care to observe and investigate CAFE's effectiveness and versatility within these applications.

First, we took an in-depth look at a pair of SAXPY applications demonstrating the implementation and performance differences between the Batch-Sync and

Subqueue Streaming methods for data transfer. As a result, we were also able to show the CAFE's flexibility through supporting different paradigms according to the needs of the developer and their application.

Second, we considered CAFE's effectiveness within a large-scale application in the form of a simple ray tracer. The multiple stages of the ray tracing pipeline allowed us to observe the powerful benefits of the manner by which our framework interfaces with the application data through the abstraction of `CafeJob`s and `CafeJobQueue`s. In addition, we also presented a set of runtime results to exhibit the superiority of Cell in a computationally intensive application.

In our third application, we demonstrated the versatility of our framework by implementing a series of image filters in accordance with the Chaining programming model. We also showed the performance benefit (in our case 1.5*x*) to employing the Chaining paradigm over the Function-Offload model for applications such as this.

Finally, through our port of the Bitonic Sort application we demonstrated CAFE's ability to preserve the same level of control and available performance granted by the hardware even through its higher level software development interface.

**Chapter 6 Notes**

1. Acceleration structures in ray tracing help to quickly determine which triangles need to be tested against the ray. As such, acceleration structures can greatly improve the performance of a ray tracer.

## Chapter 7: Future Work

Although, CAFE has undergone a few changes from its initial conception, the drive of creating a lightweight, flexible framework has remained throughout. The utilities featured in the CAFE library were added because we found them to be useful across multiple applications and they stood out as foundational components to Cell application development. In accordance with these fundamental principles, there are a few outstanding items that we would like to suggest as future work for the framework library.

While we believe our minimalist approach gives the developer the most control over the scheduling of their application, we do see the benefit in having an automatic method of ferrying a payload to and from the SPU. We believe that a simple job scheduler could be implemented as another library on top of our current base. This way, our library would still offer our clients the freedom to choose which programming model they wish to use to better fit their application needs, yet give them a more "automatic" option for those who may be less interested in the intricate control over communication. Certainly, this scheduler is a higher-level component, but, for instances where fine-grain data transfer control is not required, it would even further help relieve developers of having to implement another common mechanic to their applications.

Also, it would be an additional convenience if support were added for designating groups of SPUs by their processor IDs. This is a useful piece of information that allows clients to produce specific processor configurations such as

token rings or hypercubes. Currently through the mechanisms exposed by the CellSDK, it takes a little bit of work in order to obtain the node IDs and set up the desired configuration. However, this process appears to be fairly straightforward to generalize and could be extended into another library feature.

In the interest of porting, we think it would be useful to add a "Cell simulator" to our Win32 backend. What we mean by this is that we would like to wrap the SPU thread creation process so that depending upon which platform the code is running (either PPU or Win32) then the framework would create the appropriate thread type (SPU thread or Win32 thread). This would allow for a more uniform testing base across the Cell and Win32 environments, as well as enable us to test more of the Cell-specific code using a unit-testing framework.

Finally, although we have never required the employment of SPU overlays in our own applications, their usefulness cannot be ignored. As such, we believe that additional support for overlays could be added to ease the development efforts of anyone who wished to dynamically inject code at runtime rather than spawning new SPU threads for each task type.

## Chapter 8: Conclusion

> *"One thing it is important to note is that while Cell offers huge potential computing performance, it doesn't come free."*
>
> – Nicholas Blachford [Blachford06]

Throughout this work, we have explored many facets of the Cell processor with a particular emphasis on application development. It is clear that while the Cell presents the opportunity to gain significant improvements in performance, the additional challenges that developers must face to attain this benefit can be daunting. More specifically, to take advantage of the power of the Cell, programmers must have a deep understanding of the capabilities of the hardware.

In response, several approaches, in the form of both libraries and languages, have been proposed to alleviate the additional layer of responsibility Cell imposes at the software level. As we observed, most of these solutions exclusively follow the Function-Offload programming model by either exposing the hardware through a "remote task" abstraction or by supplying an architecture-agnostic interface. In other words, none of the prior work presents the flexibility to employ an additional programming model (such as Chaining) using the mechanisms provided in the library or language. We believe this to be detrimental to applications which would benefit from being implemented in an unsupported paradigm. Furthermore, while the frameworks presented claim responsibility for the lower level routines of partitioning and transferring data back and forth from the SPUs, we also believe that they, in turn, usurp control from the developers. This is counter to one of the fundamental

principles upon which the Cell processor was designed: to give the developer *more* control over the hardware. In consequence, we believe that there is a genuine need for a framework library which supplies a sufficient set of utilities for data-partitioning and payload transfer, while preserving lower-level control and providing the mechanisms necessary to support an array of programming models.

Therefore, in this paper we introduced our own approach: Cell Architecture Framework and Extensions (CAFE), a minimalistic framework designed to assist developers in taking advantage of the computational power provided by the Cell's SPUs without forcing a single programming model onto their applications. CAFE is founded on our measurements and observations from Chapter 4, which allowed us to recognize several key principles about Cell application development (in terms of the intricacies of both SPU threads and DMA) and integrate them into the library. Additionally, the utilities provided by the CAFE library are non-intrusive do not require additional support from a virtual machine or runtime library, which can be expensive in terms of performance and memory, respectively. As a result, we believe our framework to be a unique approach.

CAFE exposes the hardware through a "remote task" abstraction called *jobs* which can be generated to serve in either of the aforementioned Function-Offload and Chaining programming paradigms. Although the routine to ferry job payloads to and from the SPUs is encapsulated in the library, the manner in which they are dispatched to the SPUs is left in the hands of the developer. As we demonstrated in Chapter 5, job dispatching can be performed in different ways, including the Batch-Sync and

Subqueue Streaming methods, which gives developers more control over the flow of data as opposed to relying on a general scheme implemented in the framework.

However, we also observed that there are a couple of disadvantages to CAFE. First, CAFE's minimalist approach requires developers to write more source code than higher level approaches supplied in the more heavyweight frameworks like RapidMind. This was a trade-off for the additional level of control CAFE offers, which we felt was an important feature to provide, especially for the scientific computing and high-performance communities. Such control is not exposed in other frameworks in an effort to make Cell application development more similar to programming for more conventional architectures where the mechanisms for data transfer are the responsibility of the hardware. Also, we found that one of the primary disadvantages to employing an abstraction in which the data payloads are "contained" by reference (as opposed to having made a deep copy into its own buffer) is that in the case of stencil methods, it places a limit on the size of the grid we can operate on due to the size of the SPU local store. This design decision was a trade-off to save both time and memory for most other scenarios which do not require a copy.

In terms of performance, CAFE was not explicitly evaluated against the measurements of other frameworks. Instead, we demonstrated that CAFE implementations can still gain the same performance improvements that may be achieved through the low-level functions of the CellSDK. Since these routines are the foundation upon which other frameworks are built, we believe that it is possible to reach similar levels in performance shown in other frameworks by using our API.

Simply put, CAFE is a lightweight, versatile framework that presents the mechanisms for data partitioning and transfer at a higher, more intuitive level, which allows developers to focus on the details of the design and implementation of Cell applications without incurring a loss in performance.

## Appendix A: Scalable City

Scalable City is algorithmically-driven work by Sheldon Brown. It has been released in a series of installations and media across the globe.

Along the way, this research project (and ultimately the CAFE framework) started as an effort to speed up the preprocessing stage of the Scalable City pipeline (described below). It was proposed that most of this step was CPU-bound and so our approach to alleviating this bottleneck was to utilize the raw processing power provided by an IBM Cell BladeServer. Due to my prior experience with the Sony Playstation 3 (PS3) through my internship at High Moon Studios, I was chosen to spearhead this development pipeline.

### Scalable City Pipeline

At its core, the Scalable City project generates the end product via procedural methods. To achieve this result, an extensive pipeline of algorithmic techniques from graphics, image processing, and computer vision was designed and implemented. The final application is a 3D virtual environment in which a user can interact with the procedurally generated world.

The Scalable City pipeline begins with real-world satellite image data. The image is gray-scaled and used to generate a heightmap corresponding to the intensities of the pixel colors. This heightmap is then altered through a series of layered transformations and summed to produce a new heightmap.

This heightmap is used to produce a terrain for the environment, upon which the urban elements will be *grown*. We use the term *grown* because the defining elements of the city, the roads, are generated via an L-system plant development algorithm. However, before the roads are laid out, a city boundary must be established. This is achieved by employing an edge-detection filter to the heightmap image.

After a suitable boundary curve has been chosen, the system begins to fill the inner region with Archimedes spirals. These curves are spawned under a customizable rule system that governs where and how curves may spawn, as well as prevents them from crossing. Curves may only start from an existing curve (including the boundary), and so as each new curve is added to the system, more possible spawn points are available to choose from. The rule system also directs the trajectory of the spirals as they stem from their designated spawn points so that they produce smooth transitions and intersections not unlike a fern.

Once a threshold of space-filling within the boundary has been met, these curves are then used as a template for the road geometry, which includes the street as well as a pair of outlining sidewalks. Roads are paved along the paths of these curves and sewn together via intersection components stationed at the point where a "child" curve stems from a "parent" curve. These intersections come in various flavors similar to the letters "Y" and "W". Furthermore, a cul-de-sac is generated at the end of each road. Finally, the road geometry is conformed to the terrain.

**Scalable City Application**

The Scalable City application allows a user to interact with the world as well as other procedural elements. A user is embodied as a vortex of vehicles that is constructive in nature. As the vortex moves along in the environment, the roads unfold in front of them, establishing their path down the L-System pattern.

Scalable City is also populated with structure components, representing walls and roofs of suburban housing. These components are generated via stereographic computer vision techniques, which allow three-dimensional mesh data to be extracted from a pair of two-dimensional images. Once generated, these components are scattered throughout the landscape and may be picked up by the forces of the user's vortex. However, instead of leading a path of destruction, the vortex assembles these affected structure components into new housing tracts, building order from chaos.

**Scalable City and Cell**

After the foundation of CAFE's core had been developed, the Scalable City application was eventually ported to utilize the power of the Cell and demonstrated in an exhibit at the IBM booth at the SuperComputing 2007 conference in Reno.

**Appendix B: CAFE API Reference**

## Library Overview

### `cafe` Namespace:

Everything in the CAFE API exists in the `cafe` namespace so it can be used with other libraries and software without causing name conflicts. As such, clients must append their usages of CAFE types and routines with `cafe`. However, in the interest of simplicity and readability we will omit this scoping throughout this reference.

### CAFE Platform Defines:

```
CAFE_PPU
CAFE_SPU
CAFE_WIN32
```

CAFE utilizes defines to distinguish processor-specific code. We use these throughout the CAFE library implementation to keep the functionality interface sensible in the presence of the different platforms.

## CAFE Structures

In this section we outline the fundamental structures to the CAFE library.

### `MemoryRegion`:

A `MemoryRegion` is a convenient way to pass around a pointer to a contiguous region of memory accompanied by its size in bytes. The `MemoryRegion` is a data wrapper that allows CAFE routines to operate on application data without forcing it to be declared as a type from the CAFE library. This way the application and library are only coupled through this single metadata type, and the client still owns their data.

Creating a `MemoryRegion`:

```
MemoryRegion(start, sizeInBytes);
```

A `MemoryRegion` must be initialized upon creation. It takes in the start pointer to a data buffer, and the size of the data buffer in bytes.

`MemoryRegion` Members:

`start` is the starting address to the data buffer to wrap.

`sizeInBytes` stores the size of the wrapped data (in bytes).

`MemoryRegion` Operations:

None.

**Address64:**

The `Address64` typedef is a union of 64-bit address representations. It is accompanied by a pair of free functions for convenience.

`Address64` declaration:

```
typedef union Address64_TYPE
{
      unsigned long long ull;
      unsigned int ui[2];
      void* p;

} Address64;
```

An `Address64` can be accessed as a single `unsigned long long`, a pair of `unsigned int`s, or as a `void*`.

`Address64` Free Function Routines:

```
void Print(const Address64& addr64, const char* label=NULL)
```

Prints the `Address64` in hexadecimal, appending an optional label if provided.

PPU only:

```
void ReceiveAddress64FromSpu(Address64& addr,
      const speid_t spuID)
```

Receives an `Address64` sent by the `SendAddress64ToPpu()` routine on the SPU.

SPU only:

```
bool SendAddress64ToPpu(const Address64& addr)
```

Sends an `Address64` to the PPU. Returns `true` if PPU acknowledges it has correctly received the `Address64`.

## Data-Partitioning Utilities

In this section we outline the data-partitioning utilities provided in the CAFE library.

### `CafeJob`:

`CafeJob` is the core structure employed by the CAFE data-partitioning utilities (discussed later in this section). A `CafeJob` holds all of the metadata required to transfer a data payload to and from the SPU. However, it is important to note that the `CafeJob` does *not* hold the actual payload data.

`CafeJob`s are explicitly padded to have a size of 256 bytes. This is so that they may be created in sequence in an aligned buffer and be guaranteed to fall on a DMA-compliant boundary.

Queues of `CafeJob`s are generated by the `CafeJobQueueGenerator` and `CafeJob`s can be DMAed to and from the SPU via the routines supplied by the `CafeJobDmaManager`. As such, the internals of the `CafeJob` were not intended to be explicitly used by clients of the CAFE library under normal usage. However, to support the implementation of a custom data transfer pipeline we have maintained a simple, public interface.

`CafeJob` Members:

> A `CafeJob` stores the two addresses: `mToAddr` and `mFromAddr`. These are PPU addresses, where `mFromAddr` designates where the payload starts when being transferred to the PPU, and `mToAddr` designates the starting address to where the payload should be DMAed on a return trip.
>
> `mSizeInBytes` stores the size of the payload (in bytes).
>
> `mOffsetInBytes` stores offset of the real data (in bytes) in the payload from the beginning of the DMA-compliant payload.
>
> `mElementCount` stores the number of elements present in this payload according to a given stride.
>
> `mChunkCount` stores the number of DMA-sized chunks is required to transfer the entire payload. All but the last of these chunks is the maximum allowed DMA transfer size: 16 KB.

**CafeJobQueue:**

`CafeJobQueue` holds a list of `CafeJob`s. While a `CafeJob` represents a single data payload (constrained by the available space on the SPU local store), a `CafeJobQueue` represents a complete DMA-compliant partitioning of a dataset. As such, by processing the data provided by all of the `CafeJob`s in a `CafeJobQueue`, the client will have processed all of the data within the original dataset from the PPU.

The `CafeJobQueue` is responsible for the memory of the `CafeJob`s it contains, but *not* the memory of the corresponding dataset.

`CafeJobQueue`s are generated by a factory called the `CafeJobQueueGenerator`. They may be partitioned into subqueues by calling the `PartitionIntoSubqueues()` routine described below.

`CafeJobQueue` Member Functions:

```
void Init(const int jobCount)
```

Allocates the memory for the `CafeJob`s and sets the `mJobCount` member.

```
void Clear()
```

Clears the memory of the `CafeJob`s to `NULL`.

```
void PartitionIntoSubqueues(CafeJobQueue* subqueues,
      const int subqueueCount) const
```

Partitions the CafeJobQueue into the given number of subqueues, which are returned in the given array of `CafeJobQueue`s. The partitioning method used is data cyclic.

```
int GetJobCount() const
```

Returns the number of `CafeJob`s in the queue.

```
int GetElementCount() const
```

Returns the number of elements in the entire `CafeJobQueue`.

**CafeJobQueueGenerator:**

`CafeJobQueueGenerator` is a namespace with a pair of data-partitioning utility functions. Both routines produce `CafeJobQueue`s from a dataset (given in the form of a `MemoryRegion`).

`CafeJobQueueGenerator` Functions:

```
CafeJobQueue* GenerateJobQueue(const MemoryRegion& memRegion,
      const int maxJobSizeInBytes, const int strideSizeInBytes)
```

This `GenerateJobQueue()` routine takes in a dataset in the form of a `MemoryRegion` and partitions it into a number of `CafeJob`s, which each can hold up to the given constraint of `maxJobSizeInBytes` (we call this data segment a "payload"). During partitioning, payloads will respect the boundaries of objects within the data buffer via the `strideSizeInBytes` parameter, which is used to ensure no object is split amongst the jobs within the resultant queue.

```
CafeJobQueue* GenerateJobQueue(const MemoryRegion& memRegion,
      const int maxJobSizeInBytes, const int strideSizeInBytes,
      const int rowSizeInBytes, const int ghostRowCount,
      const unsigned int scratchBufferAddr)
```

This `GenerateJobQueue()` routine also takes in a dataset in the form of a `MemoryRegion` and partitions it into a number of `CafeJob`s. However, this routine is specialized for overlapping partitioning. In addition to the parameters outlined in the former version, this routine requires the size of a row of data (in bytes), the number of ghost rows it should account for, and the address to where the SPU should DMA the processed results.

This version is especially useful in stencil applications, such as image processing, fluid dynamics simulations, and connected component labeling.

## Data Transfer Utilities

In this section we outline the data transfer utilities provided in the CAFE library.

### `CafeJobDmaManager:`

`CafeJobDmaManager` is a class that provides the functionality for transferring the payload represented by a `CafeJob` to and from the SPU. The DMA routines have been split into pairs: `InitiateDma*()` and `FinalizeDma*()`, so that multi-buffering may be employed.

The `CafeJobDmaManager` stores the `CafeJob` and uses it to complete the DMA requests. This is handy if the client does need explicit control over the `CafeJob`, but is rather just interested in the payload. If the client is in fact interested in explicitly controlling or utilizing the `CafeJob`, we recommend using the routines provided in the `CafeJobDmaHelpers` namespace to accomplish the same end.

Creating a `CafeJobDmaManager`:

```
CafeJobDmaManager(const unsigned int cafeJobAddr,
      const unsigned int cafeJobTag)
```

Requires the address to the `CafeJob` to retrieve and store and a DMA tag. This DMA tag will be used throughout the life of the `CafeJobDmaManager` in all of the DMA transactions it makes.

`CafeJobDmaManager` Functions:

```
void InitiateDmaFromPpu(MemoryRegion& memRegion)
void FinalizeDmaFromPpu()
```

This pair of routines comprises a complete DMA transfer of the `CafeJob` payload from the PPU to the SPU. The `InitiateDmaFromPpu()` function sends out the asynchronous DMA requests with the DMA tag given in the `CafeJobDmaManager` constructor. These DMA requests are waited on as a group by calling the `FinalizeDmaFromPpu()` routine.

```
void InitiateDmaToPpu(MemoryRegion& memRegion)
void FinalizeDmaToPpu()
```

This pair of routines comprises a complete DMA transfer of the `CafeJob` payload from the SPU to the PPU. These work in similar fashion to the pair of routines outlined above.

**CafeJobDmaHelpers:**

`CafeJobDmaHelpers` is a namespace that provides the functionality for transferring the payload represented by a `CafeJob` to and from the SPU. The DMA routines have been split into pairs: `InitiateDma` and `FinalizeDma`, so that multi-buffering may be employed.

In fact, the `CafeJobDmaHelpers` namespace provides the exact same functionality as the `CafeJobDmaManager`. This is to give clients the option of explicitly controlling the `CafeJob`. The `CafeJobDmaManager` must keep the same `CafeJob` around until it is destroyed because it owns the `CafeJob` memory. The `CafeJobDmaHelpers` interface does not have this constraint.

As such, the `CafeJobDmaHelpers` interface provides a routine for retrieving a `CafeJob`, as well as the same payload transfer functions found in the `CafeJobDmaManager`.

`CafeJobDmaHelpers` Functions:

```
void DmaCafeJobFromPpu(CafeJob& cafeJob,
      const unsigned int cafeJobAddr,
      const unsigned int dmaTag)
```

The `DmaCafeJobFromPpu()` routine retrieves the `CafeJob` from the given PPU address using the given DMA tag. It should be noted that this routine allows the client to control the memory allocated for the `CafeJob`; however, this client-owned `CafeJob` must be created to be DMA-compliant.

```
void InitiateDmaFromPpu(MemoryRegion& memRegion,
      const CafeJob& cafeJob, const unsigned int dmaTag)
void FinalizeDmaFromPpu(const unsigned int dmaTag)
```

This pair of routines comprises a complete DMA transfer of the `CafeJob` payload from the PPU to the SPU. The `InitiateDmaFromPpu()` function sends out the asynchronous DMA requests for the chucks in the `CafeJob` with the given DMA tag. These DMA requests are waited on as a group by calling the `FinalizeDmaFromPpu()` routine with the *same* DMA tag the payload was requested with.

```
void InitiateDmaToPpu(MemoryRegion& memRegion,
      const CafeJob& cafeJob, const unsigned int dmaTag)
void FinalizeDmaToPpu(const unsigned int dmaTag)
```

This pair of routines comprises a complete DMA transfer of the `CafeJob` payload from the SPU to the PPU. These work in similar fashion to the pair of routines outlined above.

## Synchronization Utilities

In this section we outline the synchronization utilities provided in the CAFE library.

### `Barrier:`

The `Barrier` namespace contains a pair of functions (one for each processor type) that implements a PPU-coordinated barrier via the use of mailboxes.

PPU only:

**`bool SyncSpus(speid_t* speids, const int kSpuCount)`**

Coordinates the barrier across the given SPUs. Returns `false` if barrier fails to execute; an error message is printed to `stderr`.

SPU only:

**`bool SyncSpus(const int sendMsg)`**

Signals PPU this SPU has reached the barrier and waits while PPU coordinates with other SPUs. The `sendMsg` parameter is the message sent to the PPU; upon successful execution, this same message will be returned from the PPU once all SPUs have been synchronized. Returns `false` if barrier fails to execute; an error message is printed to `stderr`.

**References**


[AMD CTM] AMD Close To Metal (CTM) Technology. http://ati.amd.com/companyinfo/researcher/resources.html.

[AMD Stream] AMD Stream Processor. http://ati.amd.com/products/streamprocessor/index.html.

[ATI R600] ATI Radeon R600 (Radeon HD 2000 Series). http://ati.amd.com/products/hdseries.html.

[Balart04] Balart, J., A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, J. Labarta. *Nanos Mercurium: a Research Compiler for OpenMP*. Technical University of Catalonia. Barcelona, Spain. 2004.

[Bellens06] Bellens, Pieter, Josep M. Perez, Rosa M. Badia, Jesus Labarta. *CellSs: a Programming Model for the Cell BE Architecture*. Barcelona Supercomputing Center. Barcelona, Spain. 2006.

[Blachford06] Blachford, Nicholas. *Programming the Cell Processor*. http://www.blachford.info/computer/articles/CellProgramming1.html. 2006.

[Bouzas06] Bouzas, Brian, Robert Cooper, Jon Greene, Michael Pepe, Myra Jean Prelle. *MultiCore Framework: An API for Programming Heterogeneous Multicore Processors*. Mercury Computer Systems, Inc. 2006.

[BrookGPU] Brook Programming Language. http://graphics.stanford.edu/projects/brookgpu/.

[Buck04] Buck, Ian, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, Pat Hanrahan. *Brook for GPUs: Stream Computing on Graphics Hardware*. http://graphics.stanford.edu/papers/brookgpu/. 2004.

[Cactus] Cactus Code: A General Code for Partial Differential Equations. http://www.cactuscode.org/. 1998.

[Chombo] Chombo: Adaptive Mesh Refinement Library. http://seesar.lbl.gov/ANAG/chombo/. Lawrence Berkeley National Laboratory. 2000.

[Christopher06] Christopher, Thomas W. *Bitonic Sort*. http://www.tools-of-computing.com/tc/CS/Sorts/bitonic_sort.htm.

[Du Toit07] Du Toit, Stefanus and Michael McCool. *RapidMind: C++ Meets Multicore*. Dr. Dobb's Journal. July 2007.

[Direct3D] Direct3D, Graphics API of the Microsoft DirectX Suite. http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx. Microsoft. 2007.

[Eichenberger06] Eichenberger, A. E., J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, R. Koo. *Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture*. IBM Systems Journal, Vol 45, No 1. 2006.

[Fatahalian06] Fatahalian, Kayvon, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, Pat Hanrahan. *Sequoia: Programming the Memory Hierarchy*. ACM / IEEE Super Computing Conference. Nov. 2006.

[Gonzalez02] Gonzalez, Rafael C. and Richard E. Woods. *Digital Image Processing*. 2nd Ed. Prentice Hall. 2002.

[Gschwind00] Gschwind, Michael, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, Takeshi Yamazaki. *Synergistic Processing in Cell's Multicore Architecture*. IEEE Micro, Vol. 26, Number 2, March 2000, pp.10-24.

[Hearst07] Hearst Electronic Group. *Tool Optimizes Multiple Core Code*. http://www.electronicproducts.com/ShowPage.asp?SECTION=PRIMID=&FileName=swjh02.Jul2007.html. Hearst Business Communications, Inc. 2007.

[Hofstee05] Hofstee, H. Peter. *Introduction to the Cell Broadband Engine*. IBM Corporation. 2005.

[IBM05] *IBM Cell Broadband Engine Architecture v1.0*. August 8, 2005

[IBM06a] *IBM Cell Programming Handbook v1.0*. April 19, 2006

[IBM06b] *IBM Cell Programming Tutorial v1.1*. June 16, 2006

[IBM06c] *IBM Cell Programming Tutorial v2.0*. Dec 15, 2006

[IBM06d] *IBM CellSDK Libraries Overview and User's Guide v1.1*. June 30, 2006

[IBM06e] *IBM C/C++ Language Extensions for Cell Broadband Engine Architecture v2.2.1*. Nov 27, 2006

[IBM06f] *IBM SPE Runtime Management Library v1.1*. June 15, 2006

[IBM06g] *IBM SPE Runtime Management Library v1.2*. Dec 5, 2006

[IBM07] IBM DeveloperWorks Forum. *Barrier* Post. http://www-128.ibm.com/developerworks/forums/dw_thread.jsp?forum=739&thread=153049. February 14, 2007.

[Kahle05] Kahle, J. A., M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy. *Introduction to the Cell Multiprocessor*. IBM Journal of Research and Development, Vol 49. No 4/5, 2005, pp.589-603.

[Kleen04] Kleen, Andi. *NUMACTL*. http://www.linuxmanpages.com/man8/numactl.8.php Section: Linux Administrator's Manual. March 2004.

[Kunzman06a] Kunzman, David, Gengbin Zheng, Eric Bohm, Laxmikant V. Kale. *Charm++, Offload API, and the Cell Processor*. http://charm.cs.uiuc.edu/papers/CellPMUP06.pdf, 2006.

[Kunzman06b] Kunzman, David and Laxmikant V. Kale. *Porting Charm++ to the Cell Processor*. http://www.cs.utk.edu/~dongarra/cell2006/cell-slides/13-David-Kunzman.pdf, 2006.

[libSh06] libSh. http://www.libsh.org/. 2006.

[MATLAB] MATLAB. http://www.mathworks.com/. The MathWorks, Inc. 1994.

[MCF] Mercury MultiCore Framework. http://www.mc.com/microsites/cell/.

[Mercury05] Mercury Computer Systems, Inc. *Cell Architecture Advantages for Computationally Intensive Applications*. Mercury White Paper. 2005.

[Mercury06a] Mercury Computer Systems, Inc. *Algorithm Performance on the Cell Broadband Engine Processor*. http://www.mc.com/uploadedFiles/Cell-Perf-Simple.pdf. Presentation Slides. June 28, 2006.

[Mercury06b] Mercury Computer Systems, Inc. *MultiCore Framework: Harnessing the Performance of the Cell BE Processor*. MultiCore Plus Data Sheet. 2006.

[Moore65] Moore, Gordon E. *Cramming More Components onto Integrated Circuits*. Electronics, Vol 38, No 8, April 19, 1965.

[MPI03] MPI: Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/. 2003.

[NAMD] NAMD: Nanoscale Molecular Dymanics Simulation. http://www.ks.uiuc.edu/Research/namd. University of Illinois at Urbana-Champaign. 2007.

[OpenGL] OpenGL. http://www.opengl.org/. 1997.

[nVidia G80] nVidia G80 (GeForce 8 Series). http://www.nvidia.com/page/geforce8.html.

[PeakStream07] PeakStream Inc. *The PeakStream Platform: High Productivity Software Development for Multi-Core Processors*. Product Information Packet. 2006.

[RapidMind] RapidMind Development Platform. http://www.rapidmind.net/.

[RapidMind07a] RapidMind Inc. *Writing Applications for the Cell BE Processor Using the RapidMind™ Development Platform*. http://rapidmind.net. 2007.

[RapidMind07b] RapidMind Case Studies. http://www.rapidmind.net/case-studies.php. 2007.

[Sedgewick98] Sedgewick, Robert. *Algorithms in C++ Parts 1-4*. 3rd ed. Addison-Wesley, 1998.

[Sequoia] Sequoia Programming Language. http://sequoia.stanford.edu/.

[Shirley02] Shirley, Peter. *Fundamentals of Computer Graphics*. 1st ed. AK Peters, Ltd. 2002.

[STI05] STI Design Center. *A Remote Procedure Call Implementation for the Cell Broadband Architecture*. SCEI / Toshiba / IBM. 2005.

[Stokes06a] Stokes, Jon. *AMD Latest to Tout GPU as Stream Processor*. Ars Technica. http://arstechnica.com/news.ars/post/20061115-8230.html. Nov. 15, 2006.

[Stokes06b] Stokes, Jon. *PeakStream Unveils Multicore and CPU/GPU Programming Solution*. Ars Technica. http://arstechnica.com/news.ars/post/20060918-7763.html. Sept. 18, 2006.

[Stokes07a] Stokes, Jon. *Google buys Peakstream Inc.*. Ars Technica. http://arstechnica.com/news.ars/post/20070605-google-buys-peakstream-inc.html. June 5, 2007.

[Stokes07b] Stokes, Jon. *PeakStream Bets Farm on CPU/GPU Fusion*. Ars Technica. http://arstechnica.com/news.ars/post/20070311-peakstream-bets-farm-on-cpugpu-fusion.html. March 11, 2007.

[Titanium] Titanium. http://titanium.cs.berkeley.edu/. UC Berkeley. 1998.

[Tommesani03] Tommesani, Stefano. *Intel SSE*. http://www.tommesani.com/SSE.html. 2003.

[UPC] UPC: Unified Parallel C. http://upc.lbl.gov/. UC Berkeley. 1999.

[Williams06] Williams, Samuel, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, Katherine Yelick. *The Potential of the Cell Processor for Scientific Computing*. Lawrence Berkeley National Laboratory. 2005.

[Woo07] Woo, Mason. *PeakStream: The Software Platform for the Next Generation of HPC Applications*. Presentation Slides Handout. April 16, 2007.