Bacon: A Data Parallel Programming System with Just In Time Partial Evaluation

PhD Dissertation Proposal

Nat Tuck
University of Massachusetts Lowell

November 18, 2012
1 Motivation

Today, computers are parallel. Even cell phones have quad-core processors and highly parallel graphics processors with programmable shaders[1]. A high end commodity workstation can have both 64 CPU cores and several general purpose graphics processing units (GPGPUs) that together can perform 8192 floating point operations in parallel\(^1\). Taking advantage of this parallelism is no longer optional for computationally intensive programs.

This raises the question: how do we easily write efficient parallel programs? In a perfect world, we would be able to write our programs in a high level language like Python and have them execute with a perfect parallel speedup on modern parallel hardware. Unfortunately, most of the programming languages, libraries, and development tools that we have assume serial program execution. Many of the popular high level languages, including Python, Ruby, JavaScript and others, are designed so that no more than one thread can be executed in parallel. This makes the problem very difficult, since programmers are understandably attached to their existing programming tools and code bases. Any technique for parallelization that required throwing these out would be a hard sell.

Luckily, many programs have most of their heavy computation requirements isolated in a few relatively small regions of their code. A traditional optimization technique is to separate out these computational hot spots into their own routines, rewrite them a more efficient (usually lower level) programming language, and then call them from the main program. These rewritten hot spot routines are called “compute kernels”. This method works well for adding parallelism to a program; the kernels can be made parallel while the bulk of the program remains serial.

There are many ways to express parallelism in code, all with different benefits and drawbacks. One model that is easy to understand and maps well to a variety of hardware is array-based SPMD (single program multiple data) data parallelism. In this model, one piece of code operates on an entire (possibly multi-dimensional) array by running in parallel once for each element in the array.

The current cross-vendor standard for GPGPU programming is called OpenCL. OpenCL uses this model of writing separate SPMD kernels and executing them from a host program. The OpenCL standard[2] consists of two major pieces. First, it defines a programming language called OpenCL C for writing compute kernels to run on parallel hardware. Second, it defines runtime APIs for C and C++ that allow these kernels to be compiled, loaded, and executed from programs running on a host CPU.

OpenCL C uses the syntax of C99\(^2\) and provides a set of built in data types and functions that expose the numeric computation capabilities common to modern GPU devices. The specification explicitly disallows the use of various C99 functionality that is not supported by GPU hardware, including function pointers, recursion, and any sort of dynamic memory allocation or array sizing.

Rather than providing a stand-alone program to compile OpenCL C kernels, the OpenCL C and C++ APIs give developers the pieces necessary to build a compiler into their host program. This allows OpenCL programs to be portable across different hardware architectures by delaying compilation until runtime when the target GPU device is known. Unfortunately, it also requires each developer to write quite a bit of code to load the kernel source code and perform various other bookkeeping activities.

\(^1\)Quad socket AMD Opteron system with four Radeon 7970 GPUs

The phrase “portable assembly language” is sometimes used to describe the C programming language. OpenCL is now the portable assembly language of GPU-like parallel processors. A few years ago, when the potential speedup from parallelism on consumer devices was in the single digits, it made sense to do other optimizations first; with a factor of two parallel speedup, there’s no reason to rewrite a kernel for parallel execution before rewriting it in hand-optimized C. Until recently, high level parallel languages didn’t really make sense outside of large clusters for high performance computing.

Today that tradeoff has changed. According to the Computer Language Benchmarks Game[3], the speedup from using a fast, low level language like C over a slow high level language like Ruby is around a factor of 50. Commodity machines can have more than 50 CPU cores, and GPUs can be two orders of magnitude more parallel than that. Today a program written in language like Ruby that parallelized perfectly could actually execute faster than a similarly optimized C program.

But even once thousand-core CPUs are common we would still rather not give up a factor of 50 in performance if we can avoid it. Luckily, describing computation at a higher level actually provides more information which can be used for aggressive compiler optimization. One very aggressive optimization that maps well to data parallel kernels is partial evaluation[4] (sometimes called “value specialization”[5]), where routines written for a general case computation are transformed automatically to operate on specific input cases. Computers are fast enough now that we can do this partial evaluation at runtime based on the actual values seen by the program.

The motivation of the work for this dissertation is to explore this and other optimizations that could be applied to future high level parallel programming systems.

1.1 Research Project: The Bacon System

To improve the programming experience for OpenCL-compatible GPGPUs, I developed a prototype of Bacon, a new parallel programming system built on top of OpenCL. Bacon provides two main benefits to developers over the direct use of OpenCL. First, it provides a better user experience through syntax extensions in the Bacon C language (see Section 5.2) and automatic generation of interface code. Second, it uses an optimization technique called just-in-time (JIT) partial evaluation, a well known[6] optimization technique that is especially applicable to data parallel kernels and can speed up their execution significantly. Examples of Bacon C code are shown in Listing 1 and Listing 2. Ten new features implemented in the prototype Bacon system are marked in parentheses.

Just-in-time partial evaluation in Bacon operates by delaying compilation of a kernel until it is called with a concrete set of argument values. When a Bacon kernel is written, some integer arguments can be marked as specialization variables. When a kernel is first called with a given set of values for those arguments, a specialized version of that kernel is generated by partial evaluation with the values of those arguments held constant. This technique enables several optimizations and capabilities that could not be provided otherwise.

This dissertation will extend the work on the Bacon programming system in four ways. First, more sample Bacon kernels will be developed for testing and benchmarking. Second, the code generation component will be rewritten to use the LLVM Project[7] infrastructure. Third, additional capabilities and optimizations will be implemented. Finally, this enhanced Bacon system will be tested on a variety of examples and the performance will be compared to existing systems.
kernel
Array2D<float> // (1) return values
mat_mul(Array2D<float> aa, Array2D<float> bb)
{
    SETUP: // (2) in-kernel setup section
    global Array2D<float> cc[aa.rows, bb.cols]; // (3) parameterized types

    BODY:
    @range [cc.rows, cc.cols]; // (4) parallel range declaration

    float sum = 0.0;
    assert(aa.cols == bb.rows, // (5) error checking and handling
    "Matrices must have compatible dimensions.");
    for (int kk = 0; kk < aa.cols; ++kk) {
        sum += aa[$row, kk] * bb[kk, $col]; // (6) multi-dimensional arrays
    }
    cc[$row, $col] = sum; // (7) special index variables
    return cc; // (8) return variable declared and selected in-kernel
}

Listing 1: Naive Matrix Multiplication in Bacon C

1.2 Research Questions
In this dissertation I will attempt to answer the following primary questions:

- What speedup does JIT partial evaluation provide on data parallel kernels?
- How much does it improve the effect of various traditional optimizations (e.g., constant
  propagation, loop unrolling, function inlining)?
- What is the compile time vs. speedup trade-off for these optimizations?
- How do these effects vary across different hardware targets (e.g., CPU vs. GPU)?
- Can hardware information available at runtime help select appropriate optimizations
  and optimization parameters?

2 Hardware
There are two kinds of parallel processor that are now inexpensive and widely deployed
on commodity machines: Multi-core CPUs and general purpose graphics processing units
(GPGPUs). The enhanced Bacon system will focus on machines with these processors. This
section describes the architecture of these processors with concrete examples.

2.1 Multi-core CPUs
Currently the bulk of multi-core central processing units (CPUs) for use in commodity,
non-mobile general-purpose computers are AMD64-compatible processors produced by Intel
and AMD. There are a variety of other multi-core CPUs used in mobile devices, embedded devices, supercomputers, and for special purpose applications but this dissertation will focus on AMD64-compatible processors due to their easy availability.

As of April 2012, these AMD64-compatible multi-core CPUs are generally available with between 2 and 16 cores. Motherboards are widely available with up to four CPU sockets, allowing an off-the-shelf workstation or server to have as many as 64 CPU cores.

Modern AMD64-compatible machines share a number of characteristics. They provide a shared global memory in a cache-coherent non-uniform memory access (ccNUMA) configuration. In the current AMD64-compatible designs, each chip has a memory controller providing access to multiple channels of DDR3 RAM. Chips are connected by fast inter-processor interconnects, QuickPath for Intel or HyperTransport for AMD, in a ring or ring-of-rings arrangement. Each core has access to up to three levels of cache, with the largest cache shared across all the cores of a chip.

Sample specifications for high end Intel and AMD multi-core processors are shown in Table 1. From a programming perspective, these processors are relatively similar, with cache layout being the most obvious difference. One difference that isn’t entirely obvious from the numbers is that the AMD processor is actually made up of two chips with no shared cache. Communication between the two chips occurs over a HyperTransport link[8].

In addition to the parallelism from having multiple cores, modern CPUs provide another kind of parallelism in the form of single instruction multiple data (SIMD) registers and instructions. The latest SIMD extension to the AMD64 instruction set — supported by the latest generation of both Intel and AMD processors — is called VEX, which operates on 256-bit wide registers. This allows data parallel operations to be performed four or eight at a time for 64 and 32 bit values respectively.

### 2.2 General Purpose GPUs

Graphics processing units (GPUs) are massively parallel processor arrays that have developed largely to support computer gaming. As the demands of game graphics have increased, these GPUs have become more programmable. Current generation GPUs can run arbitrary

<table>
<thead>
<tr>
<th>Model</th>
<th>AMD Opteron 6276</th>
<th>Intel Xeon E5-2680</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Threads</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Chips</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Speed</td>
<td>2.3 GHz</td>
<td>2.7 GHz</td>
</tr>
<tr>
<td>Memory Channels</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>L1 Code</td>
<td>64kB / 2 cores</td>
<td>32kB / core</td>
</tr>
<tr>
<td>L1 Data</td>
<td>16kB / 2 cores</td>
<td>32kB / core</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2MB / 2 cores</td>
<td>256kB / core</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>8MB / chip</td>
<td>20MB / processor</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>147³</td>
<td>172</td>
</tr>
</tbody>
</table>

Table 1: Modern High End Multi-Core CPUs

---

³AMD64 is the vendor-independent 64-bit extension to Intel’s i386 32-bit ISA developed by AMD. Intel calls it Intel64, GNU calls it x86_64, and Microsoft calls it x64.

⁴The Opteron shares one Vector FPU per two cores, so this is a poor estimator of integer and branch performance.
<table>
<thead>
<tr>
<th>Model</th>
<th>AMD Radeon 7970</th>
<th>Nvidia GeForce GTX 680</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shaders</td>
<td>2048</td>
<td>1536</td>
</tr>
<tr>
<td>Clusters</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>Local RAM</td>
<td>64kB / cluster</td>
<td>48kB or 16kB / cluster</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>16kB / cluster</td>
<td>48kB or 16kB / cluster</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>512kB</td>
<td>512kB</td>
</tr>
<tr>
<td>RAM</td>
<td>3 GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>3788.8</td>
<td>3090.4</td>
</tr>
</tbody>
</table>

Table 2: Modern High End GPGPUs

compiled kernel code with only a few programmer constraints. This flexibility has allowed these devices to be used for general computation, and the use of GPUs in this way is frequently called general purpose GPU (GPGPU) computing.

A top of the line GPU can execute 2048 threads in parallel at 925 MHz. This means that it has theoretical floating point performance 20 times better than a top of the line CPU. Due to higher memory bandwidth and other factors, the speedup may be even higher on an appropriately parallel workload.

GPUs can’t quite be treated simply as multi-core CPUs with a large number of cores. Instead, GPUs are natively SPMD data parallel processors. They execute threads in batches called “work groups” of 32 or 64 threads (for Nvidia and AMD respectively) that execute the same instruction sequence in lock-step. This makes array data parallelism the simplest way to write programs for execution on this hardware. Further, there are several constraints imposed by the programming models, as discussed in Section 3.1.

3 Data Parallel Programming Tools

3.1 OpenCL

As mentioned in Section 1, OpenCL[2] (Open Computing Language) is a cross-vendor standard for data parallel programming designed primarily to target GPU devices. It was developed by a group led by Apple in 1998 to create an open standard in response to vendor-specific GPU programming APIs from Nvidia and Microsoft. Since then it has gone through two major revisions, with OpenCL 1.2 having been finalized in November of 2011.

OpenCL has two major parts: the OpenCL C programming language and the OpenCL runtime library. Kernels (routines to be run in parallel) are developed in OpenCL C and are loaded and executed at runtime by API calls to the OpenCL library from a host application. This delayed compilation allows OpenCL to be hardware independent; kernels are compiled for devices identified and selected at runtime.

The OpenCL standard further defines a “virtual parallel machine” model for programmers to target with OpenCL kernels. The intent is that this model be general enough to map to all target devices, while having enough of the specific properties of GPUs that it can efficiently target their somewhat peculiar architecture.

The OpenCL model describes how a kernel executes on a device. The kernel is executed in parallel over a 1, 2, or 3 dimensional grid-shaped index space called an NDRange. That grid is further divided up into “work groups” in a way that can be specified by the user if it is important for a given kernel. Each executing instance of the kernel (“thread”) is able
to access its index, and can thus determine what piece of the computation it is responsible for calculating.

Memory in the OpenCL model is separated into five explicitly separated address spaces. Global memory is shared across all threads in a kernel, but no guarantee is made that writes from one thread will be visible to other threads before the kernel is finished executing. Local memory is shared within a work group and can be synchronized for inter-thread communication by an explicit barrier. Private memory is accessible only to a single thread. As a general guideline, global memory is slow, shared memory is faster, and private memory is the fastest.

As an alternative to global memory, there are two additional address spaces that may provide special high speed caching: constant memory and images. Constant memory is read-only during kernel execution. Images are 2D or 3D arrays that can only store data in certain formats and are restricted such that a given image can only be read or written by a given execution of a kernel.

As mentioned in the introduction, use of the OpenCL C language that kernels are written in is subject to some major constraints, including a lack of function pointers, recursion, and any sort of dynamic memory allocation or array sizing. Further, the lack of any good method for communication between work groups during kernel execution is a significant limitation for many parallel algorithms. These restrictions seem to be slowly being relaxed with extensions and newer revisions, but it will be several years before OpenCL can be straightforwardly treated like a parallel C.

3.2 **CUDA**

CUDA [9] (originally “Compute Unified Device Architecture”) is a proprietary API for GPU programming developed by Nvidia. It is somewhat more mature than OpenCL but is otherwise very similar, with a slightly different set of features and restrictions. CUDA can only be used to develop for Nvidia GPUs.

3.3 **OpenMP**

OpenMP [10] (Open Multiprocessing) is an open standard API and syntax extension to C, C++, and FORTRAN for writing parallel code for shared memory multiprocessor machines. It is especially popular due to its low impact workflow. First, the programmer writes and tests a sequential program while keeping parallelism issues in mind. Second, the programmer annotates that program with “ pragmas” that expose parallelism to the compiler. Optimally, the program will then run with a good parallel speedup.

OpenMP is supported by a variety of compilers, including GCC, Intel’s ICC, Microsoft Visual Studio, IBM XL C/C++, and Oracle Solaris Studio.

3.4 **MPI**

MPI (Message Passing Interface) is an open standard API for parallel program execution on clusters. There are many different MPI implementations, including vendor-specific libraries optimized for specific hardware and widely portable open implementations like MPICH [11].

As shared memory is not generally available on clusters, MPI programs communicate by message passing. This scales better than shared memory but makes some otherwise efficient parallel algorithms no longer feasible to execute.
3.5 ZPL

Where all of the previous tools have been relatively low level, there are also higher level tools for parallel programming built on top of them. A good example is ZPL[12] (Z-level Programming Language), where the programmer describes operations on entire arrays or pieces of arrays rather than describing parallel computations by looking at single array elements. ZPL pioneered several parallel programming concepts, including the idea of “regions” which are very similar to OpenCL’s NDRanges.

3.6 Chapel

Another high level parallel programming language of note is Chapel[13]. This language, a conceptual descendant of ZPL, was developed at Cray as an entry in the DARPA funded High Productivity Computing Systems program. It tries to solve a larger piece of the parallel and distributed computing problem than other systems by being a “multi-resolution” language, allowing users to write both high and low level routines as part of the same program. This is supported by novel parallel abstractions like zippered iterators and distributed software transactional memory.

4 Related Work

4.1 Partial Evaluation

Partial evaluation was formalized by Futamura in 1971[4]. Given a procedure $\pi$ with parameters $c_1, \ldots, c_n, r_1, \ldots, r_m$ and values $c'_1, \ldots, c'_n$ for the first $n$ parameters, a procedure $\pi'$ specialized on those values can be generated by partially evaluating the procedure. Those parts of the procedure that depend on the known parameters can be evaluated, while the unknown variables remain unknown. Futamura used this concept to describe a method of generating compilers from interpreters by specializing the interpreter (the procedure) on an input program (the known values).

This technique is obviously useful as a program optimization, and there have been many projects using it over the last four decades. As the Bacon system performs partial evaluation on a language similar to C, specialization of C-type languages is most applicable.

An early example of partial evaluation of C programs is described by Anderson in 1991[14]. This partial evaluator for a subset of C is self-applicable, allowing it to be used for the compiler-generation technique described by Futamura. A speedup of nearly a factor of seven is shown for one test program.

A later partial evaluator for C called Tempo[15] does general ahead of time partial evaluation of C code. This system provided more consistent speedups and was used to accelerate HPC kernels. The same primary author, Consel, also did work on partial evaluation at runtime[6].

One especially notable example of partial evaluation for non-C programs is the Psycho[5] compiler for Python. This system generates specialized code for Python functions at runtime based on both the dynamic types of the Python variables and aggressive partial evaluation based on runtime values, which can later be backed off automatically if a more generic version of the routine is needed.
4.2 JIT Compilation

The most widely recognized use of just-in-time (JIT) compilation techniques is in the Java Hotspot® Virtual Machine (JVM)\[16\]. The JVM allows Java to make its famous “write once run anywhere” claim by delaying compilation to native code until runtime when the target hardware architecture is known and a JIT compiler for that architecture is available. Java goes so far as to provide different JIT compilers for different use cases. For example, the Java Client VM is tuned for responsiveness, which means it accepts somewhat less optimal code in exchange for shorter JIT pauses. In contrast, the Java Server VM makes the opposite trade-off, allowing longer JIT pauses to ensure faster execution once steady state is reached.

OpenCL\[2\] uses JIT compilation in much the same way Java does, except to support a wider range of novel hardware architectures. This is absolutely essential for its design goal of supporting GPUs, as the hardware architectures for these processors are still in flux. For example, the current AMD Radeon 7xxx GPU architecture is new this year, but OpenCL code written last year will run on it perfectly.

A final very notable example of JIT techniques is the research language Self\[17\]. This language pioneered the technique of using JIT specialization on data types to allow for very efficient execution of dynamic programming languages. Although type specialization is different from value-based partial evaluation, this is one of the most significant examples of JIT specialization providing significant performance benefits in practice, allowing a dynamic purely object-oriented language to approach within a small factor of optimized C in performance. These same techniques are used in the fast JavaScript runtimes in modern web browsers.

4.3 Similar Projects

Copperhead\[18\] is a project that allows data parallel code to be written directly in Python and execute efficiently on a GPU. Copperhead provides nested parallelism in a functional style, providing parallel versions of traditional operations like map and reduce. These functions execute about half as fast as hand-coded CUDA kernels. For kernels where this approach works and provides sufficient performance Copperhead or something like it may simply be the best option for development of GPU accelerated kernels. At the moment the main drawback to Copperhead is its dependence on the Nvidia CUDA platform, but the authors anticipate support for other back-ends (including OpenCL) in the future.

SEJITS\[19\] describes earlier work by some of the same people who work on Copperhead that explores JIT compilation on programs written using Copperhead-like style parallel embedded domain specific languages. Although the term “specialization” is used in the SEJITS paper to refer to hardware targeted JIT compilation, value-specific partial evaluation is mentioned as an optimization possibly enabled by a JIT compiling system.

5 Preliminary Work

I have developed a prototype of the Bacon system which consists of three components:

1. A language called Bacon C that provides for effective expression of data parallel kernels with specialization parameters.

2. A Bacon compiler that parses Bacon C code and does compile-time processing.
Before the three components of the prototype Bacon system are described, it is worthwhile to examine the life-cycle of a JIT partially evaluated compute kernel as compared to an
Traditional (e.g. FORTRAN Kernel) | OpenCL Kernel | Bacon Kernel
---|---|---
1. Developer writes host application (C++) and kernel (e.g. FORTRAN). | Developer writes host application (C++) and kernel (OpenCL C). | Developer writes host application (C++) and kernel (Bacon C). |
2. — | Developer writes wrapper code (C++) to load and run kernel. | Bacon compiler generates wrapper code (C++) to load and run kernel. |
3. — | — | Bacon compiler writes out serialized kernel abstract syntax tree (AST). |
4. Host application and kernel are compiled. | Host application and wrapper code are compiled. | Host application and wrapper code are compiled. |
5. Host application is run. | Host application is run. | Host application is run. |
6. — | — | Input data is read. |
7. — | — | Bacon library generates specialized OpenCL kernel. |
8. — | — | Target GPU is identified. |
9. — | Kernel is compiled for target GPU. | Kernel is compiled for target GPU. |
10. — | Input data is read. | Input data is read. |
11. Input data is read. | Kernel is executed on input data. | Kernel is executed on input data. |
12. Kernel is executed on input data. | Kernel is executed on input data. | Kernel is executed on input data. |

Table 3: Lifecycle of Bacon and OpenCL Kernels

ahead-of-time (AOT) or standard JIT kernel. A side by side comparison of these three compilation styles is laid out in the third columns of Table 3. Graphical diagrams for JIT compilation and specialization are also provided in the figures at the end of this section.

The basic technique of separating out high performance “kernels” from an application and implementing them in a separate language has been used in software development for decades. This even occurs for purely sequential programming. For example, an application written primarily in C++ may have high performance routines written in hand optimized assembly code or FORTRAN. Traditionally, the host application and kernel code are compiled into separate modules and then linked together before execution. This basic sequence is shown in the first column of Table 3.

Compiling everything before execution, or ahead-of-time (AOT) compilation, has one major downside: the target hardware is set when the application or module is compiled. Just-in-time (JIT) compilation avoids this problem by delaying compilation until the program is run on a specific machine, allowing the target hardware to be detected dynamically at runtime. OpenCL uses JIT techniques to allow the portability of kernels across the variety of compute-capable GPUs and other parallel acceleration hardware that provide support for the standard. The JIT OpenCL kernel life-cycle is shown in the second column of Table 3 and is shown graphically in Figure 1.

Bacon takes delayed compilation one step further, waiting to compile a kernel until it is actually called and the characteristics of the arguments can be examined. This allows just-in-time partial evaluation to be performed, as shown in the third column of Table 3. This process is shown in more detail in Figure 2 and Figure 3.
5.2 The Bacon C Language

Bacon C is based on OpenCL C with extensions for improved usability and to enable the automatic generation of C++ wrapper code. A sample Bacon C kernel that performs matrix multiplication and demonstrates eight of the new features of the language is shown in Listing 1.

Bacon preserves the OpenCL single program multiple data (SPMD) computation model. A kernel is executed in parallel over a 1D, 2D, or 3D range. Each executing instance of the kernel can query its position in that range to determine which part of the work it is responsible for performing. For example, the kernel in Listing 1 will be executed in parallel once for each element in the output matrix.

Each kernel is separated into SETUP and BODY sections. The SETUP section ((2) in the Listing 1) is for code that will run serially on the host processor while the BODY section contains the code to be executed in parallel. The SETUP section is primarily used to declare output arrays that can be returned to the host application and to compute the sizes of these arrays.

Unlike OpenCL C kernels, Bacon C kernels can return values with the return statement ((1) in Listing 1). Any variable of a simple type can be returned, as can any array passed as an argument or declared in the SETUP section. The return statement occurs in the kernel BODY ((8) in Listing 1) and can be selected conditionally (e.g., by an if statement), but the behavior if different threads try to return different values is undefined.

Each BODY includes an @range declaration ((4) in Listing 1) that specifies the range it will be executed over in parallel. Within the BODY, the current position in that range is held in special Bacon-specific variables named $col, $row, and $dep for the first, second, and third dimension respectively ((7) in Listing 1). The range is formatted like an array declaration, so a BODY with @range[4] will be executed 4 times in parallel with $col having the values 0, 1, 2, and 3. Since this is only a 1D range, the values of $row and $dep will both be zero in all four threads.

Bacon provides parameterized types ((3) in Listing 1) for 1D, 2D, and 3D arrays using C++-style angle bracket syntax. Both declarations and element access use a comma separated list of numbers in square brackets ((6) in Listing 1). The dimensions of these arrays can be accessed using struct-style dot notation. For example, a three by three by three array of integers called “cube” could be allocated with int cube[3,3,3]; and the width of that array could be accessed with cube.cols.

Additional error handling is provided through the assert ((5) in Listing 1) and fail keywords which will stop kernel execution and raise exceptions in the host process if triggered. A fail is triggered if execution in any thread reaches that statement, while an assert is only triggered if its associated condition is false.

Each Bacon kernel has a set of specialization variables. These fall into two categories. First, the dimensions of any arrays passed as arguments to a kernel are always specialization variables. Second, additional specialization variables can be specified explicitly by declaring arguments using the const qualifier. Whenever a kernel is called with a new set of values for its specialization variables a specialized version of that kernel is generated by partial evaluation and executed. Specialized kernels are cached for future calls with the same set of specialization values.

This specialization, in addition to providing performance benefits, allows variable sized arrays in thread-private memory as long as the array size depends only on const variables and array dimensions. Since OpenCL does not allow any form of in-kernel dynamic memory
allocation, this makes it possible for users to write kernels that would have been difficult to write using OpenCL directly.

5.3 Current Bacon System Prototype

The software component of the Bacon system consists of two pieces: the Bacon compiler and the Bacon runtime library. The compiler runs at application compile time and parses the Bacon C source, generating a C++ wrapper and a serialized abstract syntax tree (AST). The Bacon runtime library is called from the generated wrapper as the host application is running to load the AST, generate specialized OpenCL C code when a kernel is called, and run that code on the GPU using the OpenCL runtime.

The system is built using Perl and C++. The Bacon compiler parses the source code using Parse::Yapp[20], a yacc-compatible parser generator for Perl. This constructs the abstract syntax tree as a Perl data structure. The C++ wrapper is then generated by traversing this tree.

The generated C++ wrapper provides a C++ function with the kernel’s type signature that can be called from the user’s application. When this function is called, the Bacon runtime library loads the AST and traverses it to generate the specialized OpenCL code. Optimizations are performed at code generation time directly from the AST without the use of a traditional low level intermediate representation.

The two optimizations that are performed by the Bacon runtime library are constant propagation and loop unrolling. Constant propagation calculates the values of all the variables that have been marked as `const` by the programmer. If the value of any of these variables cannot be computed from the specialized arguments to the kernel, the Bacon runtime library will throw an exception. This information is used to construct a symbol table, and references to these variables are replaced with their constant integer values in the generated OpenCL code.

Loop unrolling is performed on any loops for which the iteration count and range can be determined after constant propagation. Short loops are fully unrolled. In this case, no loop is passed to the compiler at all. An example of this is shown at (10) in Listing 2. Longer loops are unrolled by some factor that evenly divides the iteration count.

This specialized and optimized OpenCL C code is then passed to the OpenCL compiler provided with the vendor SDK which will perform further optimizations on the generated code, including more aggressive constant propagation and possibly static register-load scheduling enabled by the Bacon pre-optimizations.

The implementation of Bacon is available publicly under an open source license. The current version can be downloaded from the public git repository

5.4 Matrix Multiplication

In order to evaluate the performance of the prototype Bacon system and determine if the optimizations provide real benefits, the run times of matrix multiplication kernels written in both Bacon C and OpenCL C were compared. These results are summarized in Table 4, which shows that Bacon is able to provide measurable performance improvements over similar programs written directly in OpenCL C.

Testing was performed on an AMD Radeon HD 5830 GPU. This is a mid-range GPU intended for high definition computer gaming. At the time of this writing, the card is a

\[http://code.ferrus.net/compilers/bacon.git\]
Table 4: Summary of 4k Matrix Multiplication Performance

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Device</th>
<th>Time (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCL - Naive</td>
<td>AMD Phenom II X3 720, Serial</td>
<td>11.9</td>
<td>1.0</td>
</tr>
<tr>
<td>OpenCL - Hand Vectorized</td>
<td>AMD Phenom II X3 720, 3 Cores</td>
<td>2.54</td>
<td>4.7</td>
</tr>
<tr>
<td>Bacon - Naive (Best)</td>
<td>Dual AMD Opteron 6234, 24 Cores</td>
<td>3.45</td>
<td>3.5</td>
</tr>
<tr>
<td>Bacon - Blocked (Best)</td>
<td>AMD Phenom II X3 720, 3 Cores</td>
<td>1.97</td>
<td>6.1</td>
</tr>
</tbody>
</table>

Table 5: NAS EP Benchmark Timings

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Device</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS 3.3 Reference</td>
<td>AMD Phenom II X3 720, Serial</td>
<td>26.0 s</td>
</tr>
<tr>
<td>NAS 3.3 Reference</td>
<td>AMD Phenom II X3 720, 3 Cores</td>
<td>9.1 s</td>
</tr>
<tr>
<td>NAS 3.3 Reference</td>
<td>Dual AMD Opteron 6234, 24 Cores</td>
<td>1.6 s</td>
</tr>
<tr>
<td>Bacon6</td>
<td>AMD Phenom II X3 720, 3 Cores</td>
<td>20.1 s</td>
</tr>
<tr>
<td>Bacon6</td>
<td>Dual AMD Opteron 6234, 24 Cores</td>
<td>2.5 s</td>
</tr>
<tr>
<td>Bacon6</td>
<td>AMD Radeon 5830</td>
<td>1.1 s</td>
</tr>
</tbody>
</table>

full hardware generation out of date, but still provides significantly more computational throughput than a CPU and provides a reasonable test platform for comparison of different kernels.

Four implementations of matrix multiplication were tested:

**Bacon - Naive**
A textbook implementation of parallel matrix multiplication. Shown in Listing 1.

**OpenCL - Naive**
An OpenCL implementation equivalent to the naive Bacon code.

**Bacon - Blocked**
Shown in Listing 2. This generalizes a 2D unrolling of the computation into square blocks.

**OpenCL - Hand Vectorized**
Hand unrolled to compute 4x4 blocks at once. Explicitly uses OpenCL’s native vector types. Based on a sample from the AMD OpenCL SDK.

The execution time of these four kernels was tested on randomly generated 4096x4096 matrices. Each test was performed five times and the average result was taken. The times were very consistent; most tests had a coefficient of variation under one percent.

5.5 NAS Parallel Benchmark: EP

Another sample kernel that has been tested is the NAS Embarrassingly Parallel (EP)[21] benchmark. This benchmark generates a large number of values using a deterministic seekable random number generator and then verifies that the expected numbers were generated.

---

6Bacon measurements do not include first-run specialization time, which generally takes about a second.
This is one of the NAS Parallel benchmarks published by the NASA Advanced Supercomputing division and is useful because it provides a variety of different problem sizes and allows for performance comparisons between different parallel systems.

This benchmark cannot be executed with the specified precision on the AMD Radeon 5830 test GPU because it requires full support of IEEE754 double precision floating point arithmetic, which is not provided by the AMD OpenCL runtime for Radeon 5000-series GPUs. Because of this inaccurate arithmetic, the Bacon GPU implementation of the benchmark is only correct to within one part in $10^{10}$ while the benchmark specification requires accuracy to within one part in $10^{12}$. CPU execution gives results within the mandated precision, as should execution on newer GPUs providing full support for IEEE754.

Test results for the NAS EP benchmark are shown in Table 5. These tests do not benefit from JIT specialization, so the use of Bacon shouldn’t provide a performance advantage over similar OpenCL code. Nevertheless, this benchmark provides some useful information and informative comparisons. First, it shows that Bacon can be used to implement well defined algorithms and get correct answers. Second, it shows that Bacon executed through the AMD OpenCL CPU runtime can run about half as fast as optimized FORTRAN code, which is a reasonably good result. Finally, it shows that it takes tens of CPU cores to match a single GPU on at least one standard benchmark (although that result is blurred by issues with GPU double precision support).

### 5.6 Robot Control Optimization

Work was done in the summer of 2011 with Michael McGuinness and supervised by Professor Fred Martin on using Bacon to port intensive kernels to GPU devices in order to speed up an autonomous robot control program. The results of this work were presented at the SPIE Electronic Imaging Conference in January 2012[22].

This work demonstrated the practical utility of Bacon in building simple GPU kernels and getting a significant speedup compared to CPU execution. Unfortunately, we also discovered that beating the performance of a hand-optimized sequential implementation of a complex algorithm with a parallel version can be very difficult.
Standard OpenCL JIT

Step 1: Developer Writes Code
Emacs (edit)

Host Program (C++)

C++ Wrapper Code

Parallel Kernel (OpenCL)

Step 2: C++ Code is Compiled

g++ (compile)

Host Program Binary

Step 3: Host Program Runs

Vendor OpenCL SDK (compile)

Kernel Executable Code

Host Program Executing
main()

OpenCL_JIT(...)

kernel_call(...)

exit()

Figure 1: Outline of Standard OpenCL JIT
Bacon Compile Time

Step 1: Developer Writes Code

- Emacs (edit)
- Host Program (C++)
- Parallel Kernel (Bacon)

Step 2: The Bacon Compiler

- bacon (compile)
- C++ Wrapper Code
- Kernel AST

Step 3: The C++ Compiler

- g++ (compile)
- Host Program Binary

The abstract syntax tree of the kernel is saved for use as input to the JIT code generator at runtime.

Figure 2: Outline of Bacon Compile Time
Bacon Run Time (Prototype)

Figure 3: Outline of Bacon Run Time (Prototype)
Proposed Bacon Run Time (CPU Target)

Step 4: Host Program Runs

- Host Program Executing
  - main()
  - kernel_call(...)
  - exit()

- Kernel Arguments
- Kernel AST

- Have Specialized Kernel?
  - yes
  - Specialized Kernel Executable Code
  - The enhanced Bacon system will be able to generate native code directly using LLVM.

  - no
    - Enhanced Bacon JIT compiler (generate code)

Figure 4: Outline of Bacon Run Time (Proposed A)
Proposed Bacon Run Time (GPU Target)

Figure 5: Outline of Bacon Run Time (Proposed B)
6 Proposed Work

For my doctoral dissertation I plan to evaluate the performance benefits and implementation details of just-in-time partial evaluation for data parallel compute kernels, building on the Bacon system described previously. The main focus of this work will be the construction and analysis of a new JIT code generator that performs partial evaluation using the LLVM Project\[7\] compiler toolchain.

This work will consist of four major elements. First, a new LLVM-based code generator will be built. Second, a testing framework will be constructed to allow different configurations of the code generator to be tested. Third, a set of interesting test cases will be assembled. Finally, the code generator will be tested with different combinations of optimizations, test cases, and hardware targets.

6.1 Research Questions

As mentioned in the introduction, there are several research questions that this dissertation will answer. In this section, an approach for answering each of these questions will be described.

What speedup does JIT partial evaluation provide on data parallel kernels?

A testing framework will be constructed along with a set of test kernels which can be used to measure the performance benefits of JIT partial evaluation with the new code generator. In addition to Bacon test kernels, OpenCL test kernels annotated for JIT partial evaluation will be used to expand the number of test cases available.

How much does it improve the effect of various traditional optimizations (e.g., constant propagation, loop unrolling, function inlining)?

The existing Bacon prototype relies on a proprietary vendor OpenCL compiler for the key optimizations that allow JIT partial evaluation to provide a speedup. The new LLVM-based code generator will allow these optimizations to be selected and tuned directly so as to provide the maximum performance improvement. The space of possible optimization configurations is too large to be tested exhaustively, but a reasonable subset will be explored using a combination of automated testing and educated guesses.

What is the compile time vs. speedup trade-off for these optimizations?

Code generation speed is very important in a JIT specializing compiler. Time spent compiling a new version of a kernel directly reduces the benefit of that kernel executing faster due to specialization. The existing Bacon prototype is very slow, taking nearly half a second to compile a kernel, due largely to its reliance on a opaque vendor OpenCL compiler. The new code generation backend will allow this tradeoff to be considered directly, at least when compiling kernels for CPU execution.

How do these effects vary across different hardware targets (e.g., CPU vs. GPU)?

Timings will be taken in various configurations for all of the target hardware being analyzed. Some measurements will not be meaningfully possible for GPU targets due to the use of
a proprietary vendor toolchain for kernel execution. Specifically, compile time / speedup
tradeoffs will not be analyzed for GPUs. The effects of different optimizations should be
testable on GPU targets by using the LLVM OpenCL backend and disabling some optimiza-
tions in the vendor OpenCL compiler using documented compiler flags.

Can hardware information available at runtime help select appropriate opti-
mizations and optimization parameters?

This is an area that hasn’t been explored at all in the Bacon prototype, but should be
feasible using the new LLVM code generator. Extensive hardware layout and specifications
are available for CPU targets using the Portable Hardware Locality library (hwloc[23]). This
information should be especially useful for hardware and problem size aware scheduling.

6.2 New Test Kernels

There are currently only two non-trivial kernels available for testing the Bacon system:
Matrix Multiplication and the NAS EP benchmark. This is not a broad enough set to
meaningfully compare different compiler optimizations.

Writing meaningful and comparable kernels from scratch is time consuming and error
prone, so only a few additional kernels will be translated into the Bacon language. In order
to allow for more complete testing of JIT partial evaluation with the new code generation
system it will be made to support partially evaluating annotated OpenCL kernels directly.
This will enable the use of existing OpenCL benchmarking suites to test and benchmark the
new code generator while also allowing apples-to-apples comparisons to be made between
normal and partially evaluated OpenCL kernels.

6.3 Existing OpenCL Test Kernels

Several good suites of appropriately licensed OpenCL benchmarks exist that can be drawn
from to build the Bacon testing suite. Kernels will be selected from those available with
the goal of representing a variety of different kinds of algorithm without spending too much
time on translation and integration. Kernels that are available include:

6.3.1 The SNU OpenCL NAS Parallel Benchmark Suite

The NAS parallel benchmarks from the NASA Advanced Supercomputing Division were
ported to OpenCL[24] by a team at the Center for Manycore Programming at Seoul National
University in Korea. These kernels are intended to be representative of high performance
computing in areas like physics and engineering.

- Block Tri-Diagonal Solver (BT)
- Conjugate Gradient (CG)
- Embarrassingly Parallel (EP)
- Discrete 3D Fast Fourier Transform (FT)
- Integer Sort (IS)
- Lower-Upper Gauss-Seidel Solver (LU)
• Multi-Grid (MG)
• Scalar Penta-Diagonal Solver (SP)

6.3.2 The Phoronix OpenCL Test Suite
This suite of benchmarks was compiled by the Linux hardware news and reviews website Phoronix[25] to enable them to benchmark hardware with OpenCL support.

• Mandelbrot Set Fractal
• Julia Set Fractal
• Small PT
• Mandelbrot Bulb
• Gluxmark

6.3.3 The Rodinia Benchmark Suite
The Rodinia Benchmark Suite[26] was developed at the University of Virginia and includes kernels intended to be representative of high performance computing in the areas of medical imaging, bio-informatics, and artificial intelligence.

• Leukocyte
• Heart Wall
• CFD Solver
• LU Decomposition
• Back Propagation
• Needleman-Wunsch
• Kmeans
• Breadth-First Search
• SRAD
• Streamcluster
• Particle Filter
• PathFinder
• Gaussian Elimination
• k-Nearest Neighbors
• LavaMD
6.4 LLVM JIT Code Generator

6.4.1 Overview of LLVM

LLVM[7] (the Low Level Virtual Machine) is a framework for building compilers developed at the University of Illinois. LLVM is a set of tools and C++ libraries that manipulate a common single static assignment (SSA) low level intermediate code representation. It provides various “passes”, like optimizations and code generation back-ends, that transform programs in this representation. Many of the traditional compiler optimizations as well as native code generation for most modern CPUs are provided in a form that is suitable for building a JIT compiler.

6.4.2 LLVM Code Generator

The main software development component of this dissertation will be to use LLVM to build a new code generation back end for the Bacon system. This will allow Bacon to generate code using this modern commercial grade compiler system and take advantage of all the optimization passes and tools provided by the LLVM project.

Using LLVM directly will allow Bacon to target CPUs natively without relying on any proprietary vendor OpenCL implementation. This will provide two key benefits. First, it will make lower level CPU optimizations (e.g., thread scheduling) possible. Second, it will give full control over the optimization vs. JIT time trade-off for CPU code, which will allow for compilation time amortization to be considered for kernels operating on small input data.

A new code generator will necessarily mean a new scheduler, since scheduling is currently done by the OpenCL library. This provides additional opportunities for performance tuning. Further, this will make possible support for clusters of machines, a configuration that is not currently supported via OpenCL.

It will be possible to target GPUs via LLVM as well. A LLVM backend that generates OpenCL code was recently released[27]. Using this backend, the new LLVM code generator will be able to replace the existing OpenCL code generator entirely while still allowing Bacon to generate code for any hardware with OpenCL support.

Further, direct code generation support for both AMD and Nvidia GPUs is currently being integrated into the LLVM development branch for inclusion in future versions of LLVM. When this occurs, GPU code generation will be on equal footing with code generation for CPU.

6.5 New Optimizations

Once the new test kernels and the LLVM code generator are available, it will be possible to focus on the main research question of this dissertation: What optimizations are especially effective in a just-in-time specializing data parallel compiler?

A variety of optimizations will be implemented, benchmarked, and evaluated for both generated code speedup and speedup compared to the time cost of the optimization pass for a given target. The use of hardware-specific information available at runtime to select appropriate parameters for optimization passes will be considered.

LLVM provides several optimization passes that seem promising. Some of these passes are used in the current Bacon prototype via AMD’s OpenCL SDK, but the new LLVM code generator will allow these passes to be evaluated explicitly and independently.
• Constant propagation
• Dead code elimination
• Value numbering
• Function inlining
• Jump threading
• Loop invariant code motion
• Loop unrolling
• Simple tail-call elimination

In addition to those standard LLVM optimizations, the main new optimizations that Bacon can include are types of hardware-aware scheduling.

• **Kernel scheduling** assigns threads to execution units in order to minimize kernel execution time.

• **Thread fusion** replaces several parallel threads with a loop that iterates over several items in a single thread. This saves scheduling overhead at the cost of parallelism. This is expected to provide more benefit for CPU targets than GPU targets where there is less parallelism and more scheduling overhead.

• **Resource-aware loop unrolling** selects the factor by which to unroll loops based on information about the hardware characteristics, such as register count and the size of the level 1 code cache.

• **Multi-kernel scheduling** varies execution scheduling based on resource utilization. When multiple kernels are executing in parallel there is contention for a limited supply of execution units and RAM. Depending on the properties of the kernels, sequential execution, interleaved execution, or static allocation of execution units may be the most efficient strategy.

### 6.6 Target Hardware Configurations

The enhanced version of Bacon will continue to primarily target homogeneous systems with reasonably coherent shared memory. This means single machines execution either on a single GPU or one or more multi-core CPUs. These are the two simplest configurations, and any results shown on them are likely to be widely applicable to desktop, workstation, and server grade machines over the next decade.

One additional interesting configuration is cluster computing, where computations are distributed across several computers each having several local CPU cores and its own RAM. Much of the work in the literature on parallel compilers assumes this configuration, and it would be useful to compare the Bacon system to cluster-based tools like MPI and ZPL. The main complication with this configuration is the distribution of memory across the multiple machines in the cluster. Distributed memory maps well to the baseline OpenCL memory semantics, which restrict inter-thread communication to a single work group. Unfortunately, this breaks down quickly with the addition of some OpenCL extensions such as atomic memory access. Bacon will not support these extensions when targeting clusters.
7 Summary

In this document I have proposed a dissertation studying just-in-time partial evaluation for data parallel compute kernels through a new programming system called Bacon. This system will allow me to explore the benefits and drawbacks of this technique and to discover and share the practical details of implementing such a system.

In order to complete this dissertation as proposed, the following additional work must be completed:

- A new code generator for Bacon must be written that performs JIT compilation to CPU native code.
- Several additional test kernels must be implemented, both using Bacon and by annotating OpenCL kernels with information to allow them to be JIT partially evaluated.
- Several new optimizations must be added to the new code generator.
- The resulting system must be benchmarked in a variety of configurations and its performance implications must be analyzed.
References


