Co-synthesis of FPGA-Based Application-Specific Floating Point SIMD Accelerators

Andrei Hagiescu and Weng-Fai Wong
School of Computing
National University of Singapore
Singapore
{hagiescu,wongwf}@comp.nus.edu.sg

ABSTRACT

The constant push for feature richness in mobile and embedded devices has significantly increased computational demand. However, stringent energy constraints typically remain in place. Embedding processor cores in FPGAs offers a path to having customized instruction processors that can meet the performance and energy demands. Ideally, the customization process should be automated to reduce the design effort, and indirectly the time to market. However, the automatic generation of custom extensions for *floating point* computation remains a challenge in FPGA co-design. We propose an approach for accelerating such computation via application-specific SIMD extensions. We describe an automated co-design toolchain that generates code and application-specific platform extensions that implement SIMD instructions with a parameterizable number of vector elements. The parallelism exposed by encapsulating computation in vector instructions is matched to an adjustable pool of execution units. Experiments on actual hardware show significant performance improvements. Our framework provides an important extension to the capabilities of embedded processor FPGAs which traditionally dealt with bit, integer, and low intensity floating point code, to now being able to handle vectorizable floating point computation.

Categories and Subject Descriptors

C.1.3 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles—*Adaptable architectures*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Co-synthesis, Custom instructions, SIMD

1. INTRODUCTION

Embedded applications vary widely in their code structure and profile. As they are often subjected to serious power and resource constraints, specialized hardware extensions are added very conservatively in embedded processors. Designers of such applications often use reconfigurable extensions to supplement their silicon processor [8, 22]. These extensions are particularly effective when there is a large amount of bit- or word-level parallelism. Although floating point applications often come with even higher amount of parallelism, they are generally not common in reconfigurable computing. They are deemed to consume too much resources, and the desired performance can only be obtained by hand-tuning the application, including conversion to fixed point and then tackling the precision issue. Yet, floating point computation is often the most straight-forward means of expressing an algorithm, especially when fractions and accuracy are involved. In this paper, we aim to show how parallelism in floating point code can be exploited *automatically* using a flexible co-designed approach that includes support for vector operations in a reconfigurable SIMD architecture.

A standard architecture for effective reconfigurable computing consists of processor cores coupled with reconfigurable hardware fabric that often resembles field programmable gate arrays (FPGA) [1, 19]. The reconfigurable fabric offers flexibility, but one cannot possibly hope to match the speed and efficiency of a silicon processor core. On the other hand, a silicon processor core with full vector capabilities like those found in desktop- and server-class processors would mean committing silicon without consideration for the applications' requirements. A dual approach requires a dedicated interconnect, and the combined performance is affected by the partitioning strategy and the data transfer overhead. Finer grain partitioning proves beneficial only when a fast interconnect is available (i.e. when both the processor core and the reconfigurable fabric are placed on the same die).

Embedded processors are getting faster. However, they seldom offer the complete set of support for parallel instruction decoding and issue, as well as enough arithmetic units to satisfy compute intensive applications. In particular, floating point units with long pipelines and separate register file are often not found in the embedded cores, and have only been recently introduced as specialized coprocessors [14]. This deficiency goes beyond compute intensive applications because many algorithms, such as DSP filters, are often designed first in a tool like Matlab before being implemented in an embedded setting [6]. To avoid slow software floating point emulation, designers have to carefully tune their applications to use fixed point computing instead. However, the complexity or certain characteristics of the applications may prevent the scaling of this approach. When input or key parts of the algorithm are changed, the application has

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

to be re-analyzed, often manually, for changes in precision and error propagation. In this paper, we examine the alternative of using the reconfiguration fabric to implement floating point units when the need arises [7]. In particular, we propose an automated design flow that takes advantage of existing auto-vectorization capabilities in compilers, and co-synthesizes code and customized floating point SIMD extensions in reconfigurable hardware. We also propose an algorithm that determines the optimal configurations for our SIMD architecture under the given resource constraints.

SIMD vector instructions are a natural candidate for hardware extensions [2] because they yield an efficient encoding of short instructions that capture a large number of operations and data transfers. Their regular structures also express large amounts of data parallelism. They are also flexible and allow for customization by varying the number of elements in the vector. Custom vector lengths can be used for the register set, operands and operators. We designed a novel architecture that supports the *concurrent* execution of SIMD instructions with different vector lengths. In particular, our architecture supports the concurrent execution of a mix of single precision 4-, 8-, and 16-float long vector instructions[1]. The exact mix used is determined by how the required processing throughput can be matched to the available reconfigurable resources. We achieve the optimal matching by *folding* the execution of larger vectors when resource is scarce. In essence, we expose a *set* of virtual instruction set architectures (determined by instruction level parallelism and other program characteristics) that is implemented by a shared pool of floating point execution units (determined by the reconfigurable resources available). Our framework inherits from the advantages of both traditional custom instructions and loop accelerators. We offer an alternative at an abstraction level where it is easier to find acceleration as well as resource sharing opportunities. On top of that, this approach offers a tighter integration in the design compilation flow. In summary, the major contributions of this paper are as follows.

- We present the design of a customizable SIMD floating point extension on hybrid architectures that have both embedded processor cores and a FPGA-like reconfigurable fabric.
- We implemented a co-design flow for this extension in which both the executable and the hardware for the selected configuration are automatically generated in a single pass.
- We propose a technique for further improving resource usage and energy efficiency by the independent folding of each kind of execution units in the final design.

2. MOTIVATION

In silicon-based processors, the vector instruction set and the vector length of the SIMD extension have to be chosen to suit a broad spectrum of computation patterns and instruction level parallelism exposed across all the application domains. However, if the SIMD extension is reconfigurable, then one may choose to only implement a particular set of vector instructions that best benefit the application at hand,

[1]For brevity, in the rest of the paper, we shall call these 'x4', 'x8', and 'x16', respectively.

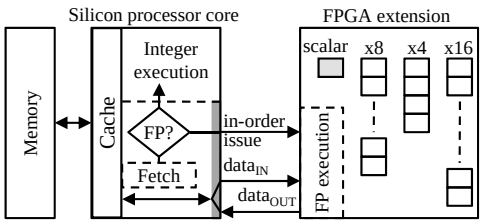

Figure 1: Our target architecture.

and have the system reconfigured to something altogether different when the demand changes. The overall system architecture we target is abstracted in Figure 1. Scalar and vector floating-point (FP) instructions are executed outside the *processor core*, in the attached FPGA *extension*. These instructions are issued in program order on a dedicated interface. One of the key insights behind our work is that unlike general integer computation, many floating point applications have the proper granularity to overcome the inherent penalty of issuing instructions outside the processor cores.

In our target system, load and store instructions have to transfer data through the processor core to the memory. Most instructions are autonomously executed by the FPGA with the exception of vector stores which require data computed in the extension to be written back to memory. The latter entails blocking the subsequent instruction issue until the data transfer is complete. Otherwise, for most other types of instructions, new instructions can be issued in consecutive clock cycles.

The following are some of the considerations that affect the selection of the vector extension to be implemented.

- The use of longer vectors will decrease the number of instructions issued to the extension, each instruction encoding computation of larger granularities.
- As the overall number of issued instructions decreases, the performance bottleneck will shift from the instruction issue to execution. There is an opportunity here to reorganize the individual operations encapsulated by each instruction, and determine a compact hardware implementation according to the exposed data dependencies.
- Larger grain computation requires more data to be transferred before computation can begin. This may cause delays especially in systems where the memory latency is large. In other words, the use of longer vectors may prevent the effective overlapping of memory transfers with computation.
- The kind of data movement is often limited by what the instructions can do. This can degrade the performance of certain operations, such as data transpositions, or the epilogue of vector reductions.

The following example shows the impact of the selected vector length on the execution time of a loop. Figure 2a shows a simple loop expressed in a C-like language based on the Altivec instruction set. The vectorization process identifies computation patterns across multiple loop iterations and coalesces them into vector operations. SIMD architectures available today mostly use vector length of four [5]. Our pseudo-compiled code example uses vector registers `vr`

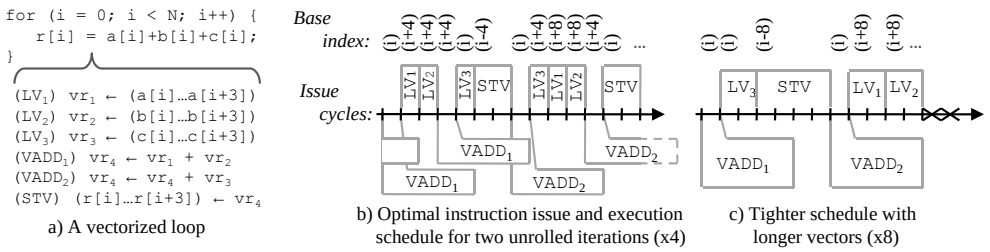

Figure 2: Executing a loop using x4 and x8 vector instructions.

that can store four single precision floating point numbers. The `LV` instructions bring the operands from memory into the registers. `VADD` instructions perform the parallel additions of the corresponding four vector elements, while the `STV` instructions store data back to memory. Because the loop is vectorized, each iteration corresponds to four scalar loop iterations.

We first analyze the issue and execution schedule of this vectorized loop as it would be handled by our target architecture with a vector length of four (i.e., x4). Figure 2b shows the issue and execution of instructions corresponding to two consecutive vector iterations processing elements with base index i and $i+4$ respectively. The compiler unrolled the loop twice and software pipelined the `LV` and `STV` instructions, placing them in adjacent cycles. Loop unrolling increases register pressure, but can partially hide the latency of the memory transfer operations and subsequently run the pipelines of the execution units more efficient. Each instruction requires a distinct issue cycle which corresponds to the data transfer between the core processor and the vector extension. Once issued, most instructions execute autonomously, spending one or more clock cycles in the pipeline of a hardware execution unit. The exception is the vector store (`STV`) instruction which occupies the issue bus for several cycles until it returnsdata from the `vr` to the processor core for subsequent stores to memory. In this example the critical path consists of the two dependent additions, $\texttt{VADD}_1$ and $\texttt{VADD}_2$. `LV` instructions are software pipelined in the available issue cycles before the start of the current iteration, while `STV` instructions are issued after the end of the current iteration.

However, due to the sequential data exchange between the core processor and the extension, repeated issuing of vector operations to the vector extension is costly. This can be detrimental to the overall performance because it limits the issue rate of compute instructions. Alternatively, the same vector loop can be compiled for a vector length of eight (x8), as shown in Figure 2c. However, in this architecture, the processor core remains unchanged, and thus all memory transfers, which are routed through the core processor, are split into chunks of four elements. Accordingly, `LV` instructions now require two issue cycles, while `STV` instructions require four cycles to transfer the operands. In this configuration, a single x8 loop iteration can handle the computation previously handled by both x4 loop iterations. The schedule becomes shorter, because less `VADD` instructions are issued.

We can compare the execution time of N iterations of an unrolled vector loop to a single equivalent iteration of a vector loop where vectors were lengthened N times. We assume that once the instructions are issued, they will execute autonomously. Ideally, performance is maximized if there is an execution schedule where, once instructions are issued, they can execute without delay caused by operand dependencies. Let t_M and t_I be the number of issue cycles used by memory transfer and non-memory transfer instructions in one iteration of the vectorized loop body, respectively. In this model, the unrolled version takes $N \cdot (t_M + t_I)$ cycles to complete, while a single iteration of the loop with longer vectors takes $N \cdot t_M + t_I$ cycles. Thus, the latter approach will improve performance by $\frac{(N-1)t_I}{N(t_M+t_I)}$. For example, if $t_I = t_M$ and $N = 4$, this translates to 37.5% improvement. In practice, this speedup is generally higher for longer vectors, because the compiler generated schedule may not be able to completely hide the instruction dependencies. However, irregular data movement patterns can make it harder to use longer vectors as additional data movement instructions will be required to correctly marshal data into the vector registers. In the final analysis, which vector length yields better performance depends on the computation pattern and available instructions.

Operations of a vector instruction may be mapped onto a reduced set of execution units by means of multiplexing, thereby trading off performance for lower resource demands. By doing so, the overall execution time may increase if the affected instructions are on the critical path. In our proposed framework, we evaluate the profitability of each vector length during compilation, and select the best based on a static model. In the resulting platform, multiple versions of the same instruction corresponding to different vector lengths may coexist. The proposed architecture allows these versions to share the execution units. Due to the sizable resources involved, the alternative of switching via runtime reconfiguration between platforms each implementing a single vector length introduces significant delays, potentially eliminating most of the performance benefits of SIMDization.

3. THE PROPOSED CO-DESIGN FLOW

An established approach for getting good performance in compute intensive applications is the use of mathematical libraries such as ATLAS [20]. The granularity and semantics of the data structures are key factors in achieving optimal results over large portions of the application. Hence, the trend to capture computation at a higher level of abstraction such as vectors or matrices. Libraries rely internally on compiler auto-vectorization of carefully written code to deliver the best performance. For our purpose, we further require support for the new vector instructions that we introduce at the hardware level.

Our choice is Eigen [9], a C++ template library for linear algebra, achieving comparable performance to ATLAS. It includes the data structures for vectors and matrices, as well

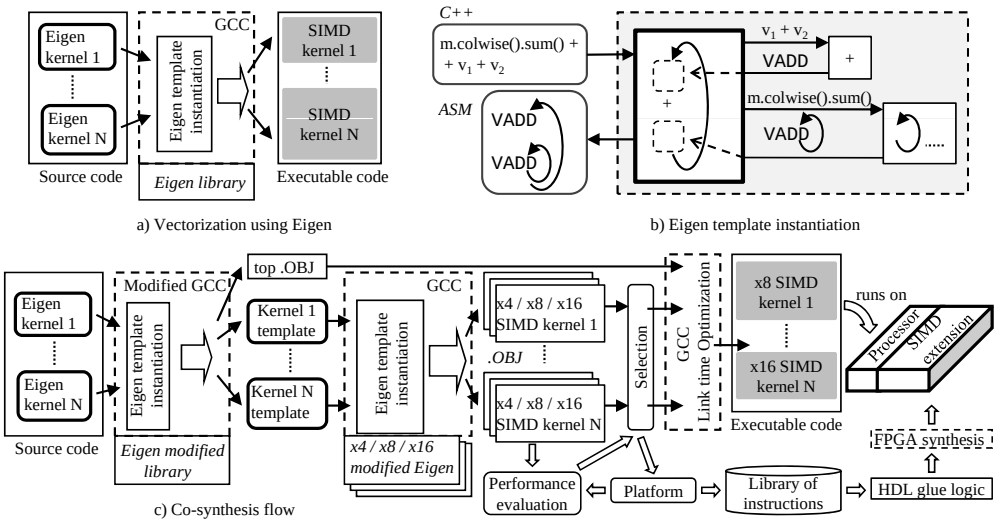

Figure 3: Our proposed vectorization flow.

as their related algorithms. Eigen uses code inlining extensively to automatically rewrite and lower the code representation, applying vectorization where possible. Its recursive template inlining and instantiation also removes unnecessary temporaries, thereby building loops with large bodies. Multiple instances of the same function template are generated and inlined in different code fragments. We shall use the term *kernel* to refer to a section of the code that has dependencies that are satisfied using (virtual) vector registers. Each kernel may consist of one or more loop nests and the code in between. What is important is that kernels are independent of each other and can be implemented using different vector lengths. The original Eigen compilation flow is presented in Figure 3a. For each computation kernel, the C++ preprocessor uses Eigen library templates and proceeds to recursive inlining which lowers the computation down to built-in functions mapping directly to assembler instructions. Internally, Eigen uses a layered instantiation and its lowest level relies on a set of primitive function templates that correspond to the actual vector instructions supported by the target architecture. The resulting executable code contains all the properly vectorized code inlined into the original functions.

The recursive template instantiation mechanism is shown in Figure 3b. The C++ computation is expressed in terms of vectors v and matrices m. The C++ code in this example sums each of the columns of m, adding the resulting vector to v_1 and v_2. The template library breaks up this sequence of operations hierarchically into a vector addition, which expects to add the result of $v_1 + v_2$ with *m.colwise().sum()*. At this point the addition is expressed in terms of abstract packets which will later be transformed to fit the length of the hardware vectors. The inlining process continues by mapping the addition of packets from v_1 and v_2 to a `VADD` instruction, while the matrix column summation is mapped to a loop that sums packets of elements from different rows. This loop and the previous `VADD` are combined in a loop nest in the final assembly code. It is only during the final mapping that the hardware vector length gets used. The packet length is propagated as a constant throughout the kernel. The compiler backend then optimises the resulting code. Recursive template instantiation and inlining are merely techniques for code rewriting. Therefore, despite the heavy use of template instantiation and inlining, the final executable does not suffer from code explosion.

In our single-pass co-synthesis flow shown in Figure 3c we have modified the library to cease automatic instantiation at the level of kernels, so that we can control and fine-tune the compilation of each template. Because kernels do not share code, we can compile each kernel independently. We pass each of these kernels to the compiler for explicit instantiation. We repeat the explicit instantiation steps so that we get all the different vector length versions of all the kernels. We project the performance of each kernel version as a function of the parameters of our SIMD extension configurations (Section 5). We also collect the list of instructions used by all versions of all the kernels. This information is used in a global selection step (Section 6) that determines the versions of each kernel to be used in the final executable, as well as the extension to be synthesized in order to execute the selected kernels. This is done in a single pass and does not require repeated hardware synthesis. We use the *link-time optimization* feature of GCC [12] to derive all versions of the kernels as well as inline the selected kernel versions for the final executable.

Eigen includes an ISA specific set of primitive template functions that corresponds to vector instructions and their GCC built-ins. We have added templates for other vector lengths in Eigen, and modified GCC by adding new built-ins and machine descriptions. It is important to note that while our choice of Eigen minimized our engineering effort, our approach can be adapted to any vectorizer that is capable of handling multiple vector lengths. This includes the GCC vectorizer, Fortran 95 vector operations, SUIF and other vectorizing compilers.

4. A CUSTOMIZED SIMD EXTENSION

In this section, we shall describe the details of the novel SIMD hardware extension that executes the vector instructions generated by our design flow. Our implementation uses the Xilinx Virtex-5 chip with its embedded PPC440 processor core. We therefore had to be compatible with the IBM AltiVec instruction set architecture [5]. However, with some

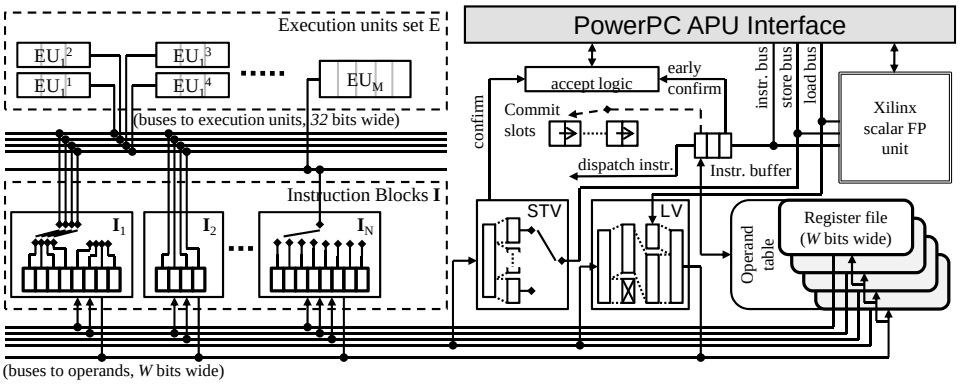

Figure 4: The architecture of the SIMD hardware extension.

amount of re-engineering, it should be possible to port this work to other similar architectures of which several alternatives are commercially available [1, 8].

The instructions for longer vector lengths are semantically simple extensions of the standard AltiVec instructions. Parallel vector operators are extended by increasing the range of their vector indexes. Several of the data movement instructions that do not use operands with absolute element indexing can also be extended in the same manner. For the instructions where an index is given as an immediate operand (i.e. `VSLDOI`, a left shift instruction), or where such indexes are included in one of the vector registers (i.e. `VPERM`, a generic permutation instruction that receives an index-based permutation pattern in a vector register) we need to cope with bit-width limitations in the index encoding. When possible, we used previously unused bits in the operand field to encode the larger index. Otherwise, we increase the granularity of the indexes, which by default is a byte. As we implement only the vector FP operations, we can increase this granularity four-fold.

The custom SIMD extension is implemented in the FPGA reconfigurable area and attaches to the Auxiliary Processor Unit (APU) interface of the embedded PowerPC processor. The overall architecture of the extension and its connections to the APU interface are presented in Figure 4. Scalar FP instructions use an IP library module provided by Xilinx, also attached to the same APU interface. The PowerPC core *issues* floating point instructions to the extention via the APU interface. Our SIMD extension includes a collection of vector FP instruction *control blocks*, a unified vector register file and a set of scalar FP pipelined execution units. An instruction control block implements one or more related vector instructions. For example, a single control block implements both the vector add and subtract instructions. The execution units are shared by the vector instructions. Instructions and execution units are connected together with minimal glue logic. We have maintained clear design boundaries between the execution units, instruction control blocks and the SIMD extension interface, allowing for a modular synthesis flow. This will also facilitate our future work to move to a run-time partial reconfiguration platform.

Assuming that there are M distinct types of execution units (i.e., adders, multipliers, fused multiply-add units), a *configuration* E is defined as the tuple $(e_1, \ldots, e_M)$ where each e_i is the number of execution units of type EU_i to be instantiated. The extension implements the instructions using N instruction control blocks $\mathbf{I} = \{\mathbf{I}_1, \ldots, \mathbf{I}_N\}$. Note that an instruction and its control block have the same vector length, denoted by $\|\mathbf{I}_k\|$, that is implicitly encoded. The pair $(E, \mathbf{I})$ fully characterizes our SIMD extension.

A single register file is used by all the vector instructions, irrespective of their vector lengths. It is configured such that it can store the longest supported vectors. Instructions handling shorter vectors will use only the lower bits of each register. The register file is implemented using the Virtex-5's block RAMs (BRAMs). To handle the longest vectors, it has to be $W = 32 \times \max_{\mathbf{I}_k \in \mathbf{I}}(\|\mathbf{I}_k\|)$ bits wide since each single precision floating point number occupies 32 bits. Altivec instructions can have up to three input and one output operands. However, the exact position of each in the instruction encoding varies. To avoid the overhead of multiplexing the possible positions to the register file, we implemented a register file with four read ports, and one write port. The amount of BRAM available in the Virtex-5 is large enough to implement four identical copies of the register file, each allowing one synchronous read and write. Read requests from different instruction operands will be serviced concurrently by the different copies of the register file. However, all the register file copies are written concurrently on any update. This ensures that all copies of the register file contain identical data, and hence consistency is enforced.

A variant of scoreboarding is employed throughout the SIMD extension. It manages instruction issue and retirement, enabling out-of-order completion. As soon as instructions appear on the APU interface, they are copied into an *instruction buffer*. A confirmation is immediately returned to the PowerPC core. The only exception is the vector store (`STV`) instruction, which needs to return data from a vector register to the PowerPC core. This instruction sends out a confirmation signal once it completes.

In the instruction buffer, a vector instruction will wait for its *dispatch* to the corresponding vector instruction control block $\mathbf{I}_k$. The instruction is dispatched only when all its operands are available in the register file, and when the hardware determines, based on the known execution time of $\mathbf{I}_k$, that the write port of the register file is available to commit the result during the clock cycle when the execution completes. These conditions together ensure that, once dispatched, instructions execute and commit their results without blocking. We keep track of when instructions will commit their results to the register file using a set of *commit slots* that correspond, in order, to reservations during future cycles to the write port of the register file. These commit slots are maintained consistent by shifting their contents to

32-bit FP execution unit	Resources LUTs	DSP48E	Frequency (MHz)	Stages
add	621	0	188	5
mul	132	2	188	3
mul-add	1101	2	152	7

Table 1: Characteristics of execution units.

the previous slot at every clock cycle.

When an instruction is dispatched to a vector instruction control block, its operands are read from the register file. The control block will latch them and execute an internal schedule that accomplishes the desired functionality using the available execution units. If there are enough execution units, all operations can be launched in the same clock cycle (such as $\mathbf{I}_2$ in Figure 4). Otherwise, a folding mechanism, described in Section 4.1, schedules the operations on the available execution units over several clock cycles. Once the execution of the instruction completes, the result is placed on the write bus and the destination register is marked as available in the operand availability table.

Some of our design decisions were driven by the idiosyncrasies of our Virtex-5 development board [15]. Vector load (`LV`) and store (`STV`) instructions are handled in a special way so as to account for the fact that the memory APU bus width is a fixed 128 bits, regardless of the vector length. The processor core performs all memory accesses including those made by the extension. Our solution for dealing with this constraint is a special *load-shift semantics* for vector load operations: consecutive loads targeting the same vector register will shift the content of the register before incorporating the incoming data in the lower bit positions. If the instructions require a vector length of 128 bits, a single load is issued, and the corresponding data is placed in the lower 128 bits of the longer vector register, while the rest of the vector register, containing shifted data, becomes irrelevant. If instructions require longer vectors, multiple 128 bit loads are issued. Each of these loads will shift the previously loaded data to more significant positions. If loads are issued in the correct order, the long vector load can be replaced by a sequence of regular loads.

Unfortunately, vector stores cannot be handled transparently. Stores may be canceled and reissued by the PowerPC due to branch mispredictions or page faults. There is therefore no easy way to check if a previous store has succeeded or not that is also compatible with our extension. To work around this issue, we implemented an explicit bank selection instruction that specifies which part of the longer vector needs to be stored via the 128 bit APU bus. The bank index is initially reset, thereby making this mechanism transparent to the x4 instructions.

The overall extension is *feasible* if the total resources (i.e., LUTs) occupied by all the instruction control blocks, execution units and other logic, including the scalar FP unit fit the resources of the FPGA. Because we implement the register file in BRAM, we freed up a significant number of additional LUTs. We use post-synthesis resource information for individual modules of our design, and account for the additional LUTs used for multiplexers, in order to derive the total requirement of our design. Besides resource constraints, the scalability of our design is also limited by the critical path of multiplexing the results back to the register file's write port via a single result bus. Nonetheless, we have successfully placed and routed extensions with as many as 32 multiplexed write sources.

We designed our own execution units, including the single-precision floating point adder, multiplier and fused multiply-adder. Their post-synthesis resource usage and performance are shown in Table 1. The multiply-adder unit fuses the two operations without the intermediary result normalization, and hence is equivalent to its standard AltiVec counterpart. Note that this does not preclude the use of other arithmetic unit designs.

4.1 Folding of SIMD operations

Folding is the mechanism used by the vector instruction control blocks to schedule the execution of vector operations on a smaller set of execution units. Our implementation currently supports folding only for vector lengths and a number of execution units that are powers of two. The folding mechanism sequences the inputs for all operations to the execution units over several clock cycles. Because the execution units are pipelined, we can launch a new set of operations each clock cycle, i.e., the *initiation interval* is one. In particular, for a vector instruction I executed by a control block $\mathbf{I}_j$, the number of consecutive cycles required to place all the operations in the e_k execution pipelines of type E_k is: $\text{fold}(I, E_k) = \|\mathbf{I}_j\|/e_k$ if the control block $\mathbf{I}_j$ requires the use of the execution units of type E_k. Otherwise, $\text{fold}(I, E_k) = 0$.

Folding requires hardware multiplexers to redirect data from several vector locations to the execution units. These multiplexers are embedded in the instruction control block and driven by state machines. The instruction control block is aware of the number of execution units available in hardware. During instruction execution, after data is fetched from the registers, a sequence of data insertions into the pipeline of the execution units is initiated. We choose to allow the instruction to drive the folding based on a run-time configuration, which leads to slightly larger (generic) multiplexers inside the vector instruction control block. Nevertheless, we believe that this is a small price to pay for design modularity and reuse.

Folding affects the rate at which the instruction control block can handle incoming instructions. If the operators are not folded, the entire instruction execution is fully pipelined, and a new instruction can be initiated every cycle. Otherwise, the instruction control block flags the execution units as busy, and this is an additional factor that may block the dispatch of the next instruction from the instruction buffer. The total execution latency of an instruction $L(I)$ is also affected by folding. This latency consists of a fixed number of clock cycles spent in the instruction control block and in the execution pipeline, and a variable number of clock cycles required to fold the instruction. The latter depends on the vector length of the instruction and the number of execution units available to it.

5. PERFORMANCE PROJECTION MODEL

A key component of our design flow is the static evaluation of design points. In this section, we shall describe the model by which we project the performance of each kernel. Without such a model, it would be impossible to offer an automated *single-pass* design flow that selects the best possible hardware configuration without trial synthesis of many candidate configurations.

The SIMD extension described in the previous section has

two degrees of flexibility. We can adjust the number of execution units of each type. We can also choose whether or not to implement instructions of various vector lengths. Recall that while a kernel can only be of one vector length, different kernels in a single application are allowed to have different vector lengths. We can estimate the performance of each kernel on a configuration E and use this metric to drive the instruction selection and implementation in Section 6.

We start by identifying the sequence of vector instructions $I = \{I_1, \ldots, I_n\}$ for each loop body in the kernel k. For most practical situations, there is usually only one loop body in the kernel. These instructions will be issued by the PowerPC processor in program order to the SIMD extension. We assume that the remaining scalar instructions execute out-of-order and have no impact on the execution time. Furthermore, the PowerPC processor is able to start prefetching the data for all the vector loads as soon as they are encountered. The PowerPC core maintains a look ahead window of δ instructions, prefetching additional instructions while it attempts to issue an instruction to the SIMD extension. In our performance model we assumed that memory accesses will hit the cache and that a 128-bit load or store transaction takes d cycles, based on the processor memory bandwidth. We also determine the execution time $L(I_p)$ of instruction I_p. Any folding will also be accounted for in $L(I_p)$ as previously described.

Based on the above assumptions, we estimate the number of clock cycles $T(E, k)$ required to execute one iteration of a loop in kernel k on a configuration E using Algorithm 1. For each instruction I_p, we derive the following timings relative to the beginning of the iteration: (a) the time when the instruction reaches the look-ahead window (α_p), (b) the time when it is issued to the SIMD extension (β_p), and (c) the time when it finishes execution (γ_p). We also track the time when the memory bus becomes available (Ξ), and the time when execution units of each type i are available (F_i). The timing obtained at the end of an iteration is used to seed the computation of the next iteration. We then iterate till we reach a fixed point, and return that as the result. In the description, 'in(I_p)' are the predecessor instructions of I_p in the data dependency graph, and 'fold(I_p, E)' was defined in the previous section.

Let V be a set of vector lengths. In our current context, $V = \{4, 8, 16\}$. The different versions of the kernel k_x are denoted by the set $\{k_x^{v_i}\}$, $v_i \in V$. The Eigen library is modified so that we can obtain the relative iteration counts of the kernel loops compiled for each of the vector lengths. If a kernel has more than one loop, then we apply Algorithm 1 to each loop inside the kernel. We derive a combined per-iteration execution time for each kernel by summing the per-iteration execution times of each loop weighed by their relative counts. We also combine the relative counts of different kernels with actual profiling data from a scalar execution of the application to project the normalized weight $\omega_x^{v_i}$ of each vectorized kernel $k_x^{v_i}$. Profiling needs to be done only once for the non-vectorized application and can be accomplished with a regular GCC compiler and `gprof`.

The performance estimate $T(E, k)$ has to be recomputed for all kernels over all the configurations as the latency of the instructions varies as a function of the folding factor. Even though we can reuse some of the estimations, the number of design points and combination of kernels is large. In our experimental setup, for example, we had 25 configurations, and 36 instruction candidates. We can prune the exploration space based on the timing relationships between the configurations. Suppose there are two configurations, $E_1 = \{e_1^1, \ldots, e_M^1\}$ and $E_2 = \{e_1^2, \ldots, e_M^2\}$. For a configuration E_i, the minimum overall execution time $\mathcal{T}(E_i)$ is reached when the version selected for each kernel k_x has the lowest execution time. In other words, $\mathcal{T}(E_i) = \sum_x \min_{v_i \in V}(\omega_x^{v_i} \times T(E_i, k_x^{v_i}))$. However, this lower bound on execution time may not be achieved if some of the required instructions are not accommodated by the resource constraint.

Algorithm 1 $T(E, k)$ for a single loop in a kernel

Require: A configuration $E = (e_1, \ldots, e_M)$ and the sequence of vector instructions $I = \{I_1, \ldots, I_n\}$ that forms the loop body inside kernel k

$\Xi = F_1 = \ldots F_n = -\infty$
// These hold the previous iteration's issue times:
$\beta_{-\delta} = \ldots = \beta_{-1} = -\infty$
repeat
  for $I_p \in I$ **do**
    $\alpha_p = \beta_{p-\delta} + 1$ // models PowerPC lookahead
    $t = \max(0, \beta_{p-1} + 1, \alpha_p, \max_{m \in \text{in}(I_p)} \gamma_m)$
    $\beta_p = \max_{E_i \text{ used by } I_p}(t, (F_i))$; // models blocking
    for functional unit type i used by I_p **do**
      $F_i = \beta_p + \text{fold}(I_p, E_i)$
    if I_p is memory transfer **then**
      $\Xi = \max(\Xi, \alpha_p) + d$
      $\beta_p = \max(\beta_p, \Xi)$
    $\gamma_p = \beta_p + L(I_p)$ // instruction ready time
  $\tau = \gamma_n$ // ready time of last instruction
  for $j < \delta$ **do**
    $\beta_{-j} = \beta_{n-1-j} - \tau$
  for E_i **do**
    $F_i = F_i - \tau$
  $\Xi = \Xi - \tau$
until τ does not increase
return τ

6. EXTENSION SELECTION

The selection of the best extension (shown in Algorithm 2) is based on statically projecting the performance achievable by the entire application on the feasible extension configurations. As its input, it takes the independently vectorized kernel versions of the application. It also requires the set of possible vector lengths, the relative weights of the kernels in the execution time, and a resource constraint.

The output of the algorithm consists of a recommended configuration and the subset of our extended AltiVec instruction set to finally instantiate. In particular, for the latter, suppose there are N distinct instructions control blocks in the set $\mathbf{I}$, and that the set of possible vector lengths is V. The algorithm outputs a Boolean decision matrix $\Phi = \{\{\phi_1^{v_1}, \ldots, \phi_N^{v_1}\}, \ldots, \{\phi_1^{v_{|V|}}, \ldots, \phi_N^{v_{|V|}}\}\}$. $\phi_j^{v_i} = 1$ if the control block of instruction j for vector length v_i is to be supported in hardware. The resource usage of this instruction is denoted by $a_j^{v_i}$.

The design space is explored using a *dominance relationship*. Let $E_1 = (e_1^1, \ldots, e_M^1)$ and $E_2 = (e_1^2, \ldots, e_M^2)$ be two configurations. We say that E_1 is dominated by E_2 (denoted as $E1 \prec E_2$) if $\exists i, e_i^1 < e_i^2$ and $\forall j, j \neq i, e_j^1 \leq e_j^2$. In other

Algorithm 2 SIMD extension selection

Require: A set of vector lengths (V), all kernels vectorized by the various vector lengths ($K = \{k_j^{v_i}\}, v_i \in V$), the relative weight of all kernels ($\{\omega_j^{v_i}\}, k_j^{v_i} \in K$), and a resource constraint (A)

Output A configuration (Π), and the subset of instructions to be implemented (Φ)

1: $SFU = \{E | \nexists E', E \prec E' \wedge \text{res}(E) \leq A\}$;
2: $\hat{R} = \infty; \Pi = \Phi = \emptyset$;
3: **while** $SFU \neq \emptyset$ **do**
4: $E = \text{pop}(SFU)$;
5: $\mathcal{T}(E) = \sum_j \min_{v_i \in V}(\omega_j^v \times T(E, k_j^{v_i}))$
6: **if** $\mathcal{T}(E) < \hat{R}$ **then**
7: $\Phi' = \{\{\phi_1^{v_1}, \ldots, \phi_N^{v_1}\}, \ldots, \{\phi_1^{v_{|V|}}, \ldots, \phi_N^{v_{|V|}}\}\}$.
8: **SATslv minimize** $R = \sum_{j, v_i \in V} (T(E, k_j^{v_i}) \cdot \omega_j^{v_i} \cdot s_j^{v_i})$
subject to
9: $\sum_{j, v_i \in V} \phi_j^{v_i} \cdot a_j^{v_i} \leq A - \text{res}(E)$
10: $\forall j \forall v_i \in V, s_j^{v_i} \leq \phi_x^{v_i}$ if instruction x of length v_i is needed in the implementation of kernel $k_j^{v_i}$.
11: $\forall j, \sum_{v_i \in V} s_j^{v_i} = 1$
12: **if** $R < \hat{R}$ **then**
13: $\Pi = E; \Phi = \Phi'; \hat{R} = R$;
14: **if** $R > \mathcal{T}(E)$ **then**
15: $SFU = SFU \cup \{E' | E' \prec E \wedge \nexists E'' \text{s.t.} E'' \prec E' \wedge E'' \prec E\}$;
return Π and Φ

words, configuration E_1 has strictly less number of units of type i than E_2, and at most the same number of units as E_2 for all other types. In a dominated configuration, i.e. E_1 here, one may implement even more vector instructions using the difference in the resource of E_2 and E_1. Even so, we note that $E_1 \prec E_2 \Rightarrow \mathcal{T}(E_1) \geq \mathcal{T}(E_2)$ regardless of what instructions are added to E_1. This means that if the current configuration is E, and $\mathcal{T}(E)$ is larger than the best found so far ($\hat{R}$ in Algorithm 2), then none of the configurations dominated by E can do better, and so can be discarded.

Algorithm 2 starts with considering the configurations that use less resources than the given resource constraint, and are not dominated by any other configuration (line 1), one at a time. For each configuration E being explored, we compute $\mathcal{T}(E)$ by selecting the best version of each kernel without any resource constraints (line 5). If this unconstrained lower bound (i.e., $\mathcal{T}(E)$) is no better than what has been already found, we discard E as well as all the configurations dominated by E, and proceed to the next candidate. Otherwise, E is a possible solution. A set of resource-based constraints (lines 9-11) is built, and we invoke a SAT solver to generate feasible solutions with the help of an evolutionary optimizer [17] with the goal of deriving a solution with the minimum execution time. The binary decision variable $s_j^{v_i}$ indicates whether kernel j vectorized with length v_i is part of the solution. $\phi_j^{v_i}$ indicates if instruction j with vector length v_i is to be part of the final extension ISA. The constraint in line 10 is to ensure that if a particular vectorized kernel is chosen, then all the instructions used by the kernel are also chosen. The auxiliary function 'res(E)' estimates the resources used by configuration E. If the SAT

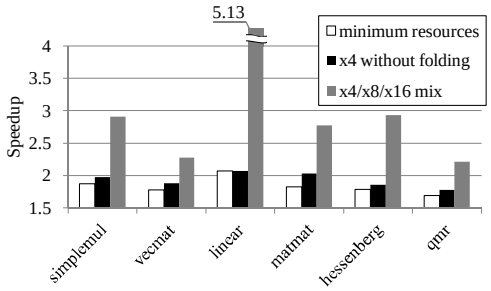

Figure 5: Speedup of different design points compared to scalar FP execution.

solver is able to arrive at a solution R that is faster than the existing best solution ($\hat{R}$) (line 12), then the newly found solution replaces it (line 13). Furthermore, if R is worse than the unconstrained bound of $\mathcal{T}(E)$ (line 14), it would imply that there is room for improvement. We will then add all configurations immediately dominated by E to the list of configurations to be considered (line 15). The idea here is that in one of these (say E'), it may be possible to obtain an improved execution time by implementing additional instructions using the resource difference between E and E'.

The algorithm is guaranteed to terminate because (a) we limit the number of iterations of the optimizer, and (b) only smaller configurations are added for future consideration. In practice, for the benchmarks reported in Section 7, it took no more than a minute on a Intel Core 2.

The solution returned by the algorithm is fed into an application of ours that puts together a mix of Verilog and VHDL modules that is then pushed through the Xilinx synthesis flow to obtain the bitstream of the SIMD extension. The solution is also used in our modified version of the GNU assembler which is utilized in conjunction with our modified GCC-LTO compiler to generate the executable.

7. RESULTS

We implemented our extensions on a Xilinx ML510 system [15]. The Virtex-5 VFX130 FPGA on board includes a PowerPC 440 core, and 81,920 look-up tables (LUTs). All the experiments reported here are based on the HDL code generated automatically by our toolchain that consists of our modified versions of Eigen and GCC. 18.5% of the LUTs were used for a system wrapper, a scalar FP unit from the Xilinx library and the SIMD extension interface. We selected a set of linear algebra benchmarks that would be ideal candidates for vectorization in embedded applications such as media processing [16], sensor array data processing, global positioning systems and beamforming solutions [14]. Several vector and matrix benchmarks are provided in Eigen. These benchmarks are compute intensive functions and reach performance comparable to BLAS on standard architectures. Furthermore, we used Eigen to vectorize benchmarks from the Iterative Template Library [13], which provides iterative methods for solving linear systems. We explored an extensive range of design points for the SIMD extension, allocating up to 67% additional LUTs. For all these points we were able to synthesize extensions with the same frequency constraints. The core processor runs at 400 MHz, while the extension runs at 133 MHz.

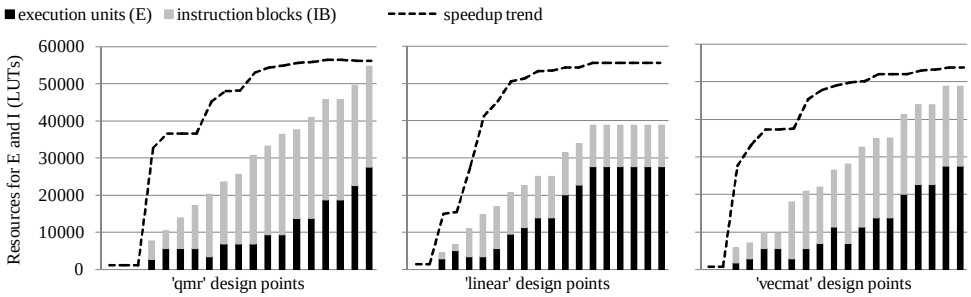

Figure 6: Resources used by execution units vs. instructions throughout the design space.

Figure 5 shows the speedup achieved by our co-synthesized design compared to scalar FP execution using the Xilinx IP core. We present the performance for three design points for each benchmark: (1) a design that utilizes a very low amount of resources, hence the execution units and instructions are constrained to 10% of the total LUTs, (2) an x4 mapping where the number of execution units matches the vector length (no folding, would correspond to a naive implementation), and (3) a design using the optimal mix of x4/x8/x16 instructions, constrained only by the maximum available resources. The minor performance improvement observed between the former two designs, which use x4 instructions, supports the observation that, for short vectors, the bottleneck is at instruction issue, and merely adding execution units has a limited impact on performance. Instead, using longer vectors led to up to 5.13× improvement.

There is a non-trivial balance between the execution units configuration E, and the instructions control blocks implemented $\mathbf{I}$. x16 instructions encapsulate larger multiplexers and hence use significantly more resources than the shorter vector versions, but deliver better performance. However, there are numerous cases where, due to resource constraints, the optimal solution involves using a smaller set of possibly shorter vector instructions complemented by a higher number of execution units. In Figure 6 we present performance and resource utilization for three benchmarks, for a set of design points when the resource constraints are relaxed. This figure presents only the total amount of resources used for each of the two configurable portions of the design (E, $\mathbf{I}$). While performance increases monotonically, the non-trivial distribution of resources between the instructions and the execution units shows the need for our search algorithm. In addition, Figure 7 gives the composition of the instructions implemented for 'qmr' at the given design points. It shows the resource usage associated with instructions of different vector lengths. x4 instructions use significantly less resources than x16.

Beside speedup, using vector instructions also leads to significant energy savings. We compare the total energy of the fastest design to that of a design using solely the scalar FP Xilinx IP core. The result reflects the total energy consumption of the FPGA core, which includes the energy of the PowerPC embedded processor. We used the current sensor provided on the ML510 board and measured its average value during program execution using a multimeter. Table 2 reports the energy consumption derived from our measurements using the best mix of instructions. The energy consumption of the best mix can be as low as 22% of the original scalar version.

8. RELATED WORK

There are a number of customized vector processor architectures [8, 21, 23] that have been proposed or are available on the market. They provide a rich set of reconfigurable parameters. However, the final result is a monoltihic processor instance with all instructions tightly integrated into the base pipeline. As such, resulting processors are implemented exclusively either in silicon or as soft-cores.

A more modular approach can be found in the sizable body of prior work on *custom instructions*, including a number of commercial products such as those developed by Tensilica [8] and Stretch [19]. They also provide SIMD custom instructions for integer applications. The typical approach used in custom instructions involves the analysis of data flow graphs obtained as a result of compilation, followed by the selection of dataflow subgraphs as candidates for custom instructions implementation [4, 24]. While the selected subgraphs are identified during compilation, platform-dependent optimizations that simplify or share resources are applied during hardware mapping. This phased approach prevents compiler optimizations from fully taking advantage of resource sharing opportunities. This is particularly important for floating point computation because the number of execution units that can be implemented in hardware is small, and a co-optimization approach can potentially yield better designs.

As shown in Section 2, issuing the instructions to the hardware extension leads to significant overhead. Specialized hardware *loop accelerators* [18, 25] have been proposed, relieving the processor of the permanent issue of instructions and operands. Using this approach, loop specific optimizations such as unrolling and pipelining can be used to improve the efficiency of the execution units. Several tools [3, 10] exist that are capable of deriving dedicated loop accelerators from source code applying static transformations to extract the necessary parallelism. This approach, however, does not support irregular loop structures or complex control flow. Also, dedicated memory connections are required to provide data for the loops. Our approach, on the other hand, relies on the core processor to resolve all control flow issues and data transfers, issuing scheduled vector instructions and operands in the proper order to the hardware.

Lastly, some vendors already offered SIMD floating point coprocessors for their embedded processors [14]. This is evidence for its growing importance in the embedded space. The iPhone, for example, includes such a core [11]. However, these offerings are silicon-based, and hence do not possess the flexibility of our solution.

	Benchmarks					
	simplemul	vecmat	linear	matmat	hessenberg	qmr
data size	256	128	1024	128	128	1024
Best time (sec)	2.45	3.11	2.42	1.45	2.22	14.98
Best energy (joule)	7.5	9.1	7.6	5.7	8.2	42.2
Scalar energy (joule)	18.4	18.3	34.1	13.9	14.3	83.4
Best / Scalar energy ratio	41%	50%	22%	41%	57%	51%

Table 2: Execution time and energy.

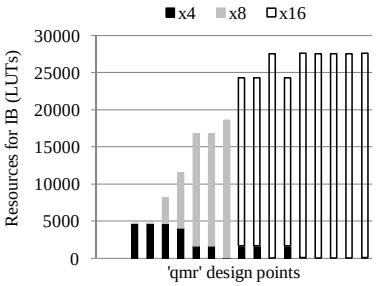

Figure 7: Distribution of resources among x4, x8 and x16 instructions for 'qmr'.

9. CONCLUSION

In this paper, we have examined the issues involved in the acceleration of floating-point computation on a hybrid reconfigurable architecture consisting of standard (integer) processor cores and a FPGA-like reconfigurable fabric. We observed that obtaining good performance for compute intensive applications on such extensions depends on a number of issues including the amount of parallelism available in the application, the structure of the loops, the processor cores' issue rate, memory bandwidth, and the reconfigurable resources available. Due to the intricate balances involved, we found that a 'one size fits all' approach to SIMDization is not always optimal. In fact, we found that for a SIMDizable application, we require a mix of vector lengths. Furthermore, this mix differs from application to application.

Based on the above insights, we built what we believe is the first fully automated toolchain that co-optimizes and co-synthesizes an application and its custom floating point SIMD extension. The toolchain leverages the latest compiler techniques in SIMDization and link-time optimization, and determines the best vector length mix for each individual kernel in a single pass, requiring only hotspot profiling.

The novel SIMD extension architecture proposed for the toolchain is able to share SIMD execution units, so as to meet a given resource constraint. Our co-synthesis toolchain has been implemented on a Xilinx Virtex-5 board. The output of the experiments were checked for correctness and the results reported are actual measurements. The experiments showed that our approach yielded up to 5.13× speedup when compared to the use of the standard Xilinx floating point IP cores. This also translated into an energy consumption that is as low as 22% of the scalar execution. As future work, we would like to examine how *partial* reconfiguration and the fusing of operations can yield even better designs.

10. REFERENCES

[1] Nios 2 processor. http://www.altera.com/literature/lit-nio2.jsp.

[2] M. O. Cheema and O. Hammami. Application-specific SIMD synthesis for reconfigurable architectures. *Microproc. and Microsys.*, 30(6):398 – 412, 2006.

[3] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *ISCA*, 2008.

[4] N. T. Clark, H. Zhong, and S. A. Mahlke. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, 54:1258–1270, 2005.

[5] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, 20:85–95, 2000.

[6] From Matlab to Embedded C. www.mathworks.com/products/featured/embeddedmatlab.

[7] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: letting applications define architectures. In *MICRO*, pages 324–335, 1996.

[8] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[9] G. Guennebaud and B. Jacob. Eigen library. http://eigen.tuxfamily.org.

[10] Z. Guo, W. Najjar, and B. Buyukkurt. Efficient hardware code generation for FPGAs. *ACM Trans. Archit. Code Optim.*, 5(1):1–26, 2008.

[11] VFPmathLibrary. http://code.google.com/p/vfpmathlibrary/.

[12] Link time optimization. http://gcc.gnu.org/projects/lto/lto.pdf.

[13] A. Lumsdaine, L.-Q. Lee, and J. Siek. The iterative template library. http://osl.iu.edu/research/itl.

[14] D. Lutz and C. Hinds. Accelerating floating-point 3D graphics for vector microprocessors. In *Signals, Systems and Computers,*, volume 1, pages 355 – 359, 2003.

[15] Virtex-5 ML510 development platform. http://xilinx.com/support/documentation/ml510.htm.

[16] V. Moya, C. González, J. Roca, A. Fernández, and R. Espasa. A single (unified) shader GPU microarchitecture for embedded systems. *HiPEAC*, 2005.

[17] Opt4J meta-heuristic optimization framework for Java. http://www.opt4j.org.

[18] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *J. VLSI Signal Process. Syst.*, 31(2):127–142, 2002.

[19] Stretch: Software reconfigurable processors. http://www.stretchinc.com.

[20] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Exp.*, 35(2):101–121, 2005.

[21] M. Woh, Y. Lin, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, R. Bruce, D. Kershaw, A. Reid, M. Wilder, and K. Flautner. From SODA to scotch: The evolution of a wireless baseband proc. In *MICRO*, pages 152–163, 2008.

[22] N. Woods. Integrating FPGAs in high-performance computing: the architecture and implementation perspective. In *FPGA*, pages 132–132, 2007.

[23] P. Yiannacouras, J. G. Steffan, and J. Rose. VESPA: portable, scalable, and flexible FPGA-based vector processors. In *CASES*, pages 61–70, 2008.

[24] P. Yu and T. Mitra. Scalable custom instructions identification for instruction-set extensible processors. In *CASES*, pages 69–78, 2004.

[25] M. Zuluaga, T. Kluter, P. Brisk, N. Topham, and P. Ienne. Introducing control-flow inclusion to support pipelining in custom instruction set extensions. *Symp. on App. Specific Processors*, pages 114–121, 2009.