Automatic Generation Of Application-Specific Accelerators for FPGAs from Python Loop Nests

David Sheffield
Michael Anderson
Kurt Keutzer

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-203
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-203.html

October 23, 2012

AUTOMATIC GENERATION OF APPLICATION-SPECIFIC ACCELERATORS FOR FPGAS FROM PYTHON LOOP NESTS

David Sheffield, Michael Anderson, Kurt Keutzer

UC Berkeley: Department of Electrical Engineering and Computer Sciences
Berkeley, CA USA
{dsheffie,mjanders,keutzer}@eecs.berkeley.edu

ABSTRACT

We present *Three Fingered Jack*, a highly productive approach to mapping vectorizable applications to the FPGA. Our system applies traditional dependence analysis and reordering transformations to a restricted set of Python loop nests. It does this to uncover parallelism and divide computation between multiple parallel processing elements (PEs) that are automatically generated through high-level synthesis of the optimized loop body. Design space exploration on the FPGA proceeds by varying the number of PEs in the system. Over four benchmark kernels, our system achieves 3× to 6× relative to soft-core C performance.

1 Introduction

The emergence of SoCs with tightly coupled FPGA fabric and high-performance multicore CPUs encourages a new way of building FPGA-accelerated systems. The FPGA + multiprocessor SoC will allow the acceleration of select kernels on the FPGA with lower overhead than previously possible. Portions of the program that cannot be easily accelerated on FPGA still run with respectable efficiency and speed on the high-performance CPUs. This motivates a selective and embedded approach to design: the programmer *selects* only certain computations for acceleration. These computations are *embedded* as a subset of a high-level language. This enables the designer to use the same source code to target both CPU and FPGA.

Such a system has benefits for both end-users and tool-writers. Users take advantage of the productivity provided by execution of non-performance critical software on a multiprocessor CPUs. They also gain performance and energy-efficiency benefits of the FPGA fabric for essential kernels through hardware acceleration. For the tool-writer, targeting only a subset of the high-level language enables great flexibility and optimization in high-level synthesis (HLS). Conventional HLS systems often target C-like languages and must support at least a subset of the C programming language semantics. This often requires handling pointer aliasing and other unsavory features of the C language. In contrast, a selective compiler can be restricted to constructs that can be handled efficiently.

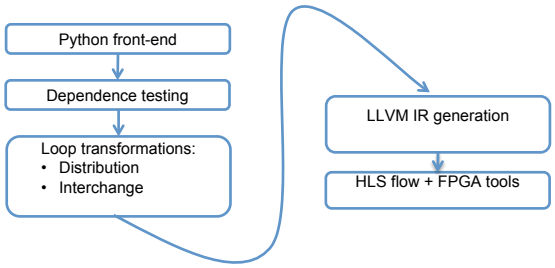


Fig. 1: Our compiler flow

We present *Three Fingered Jack*[1], a vectorizing compiler and HLS system embedded in the Python language. In our system, the programmer selects dense loop nests in Python using the ”decorator” syntax that redirects the Python runtime to our compiler. Because our compiler is restricted to dense loop nests, we can apply vectorizing compiler algorithms to the loop nests and traditional HLS techniques to automatically generate parallel processing elements.

Our approach has productivity, portability, and efficiency benefits. Portability is guaranteed as all code is valid Python. If a loop-nest cannot be compiled for the FPGA, it remains valid Python and will be executed on the CPU. We demonstrate the efficiency of our system by achieving 3× to 6× the performance of a soft-core CPU on our benchmarks. The benchmark kernels in Section 4 exemplify the productivity benefits of our system as each kernel is less than 10 lines of Python. We estimate a manual RTL implementation would be at least 20× more code.

Our work is inspired by the Selective Embedded Just-in-time Specialization (SEJITS) methodology, which uses embedded domain-specific languages (EDSLs) to help mainstream programmers target Nvidia GPUs and multiprocessor CPUs [4]. We extend the SEJITS ideas to target FPGAs. Additionally, our compiler is extensible and can support multiple backends. Though they are not the focus of this work, we also have a CPU backend and limited support

[1] `http://www.eecs.berkeley.edu/~dsheffie/threeFingeredJack`

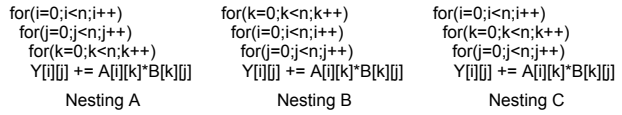


Fig. 2*:* Examples of matrix multiply loop interchange

for GPUs with an OpenCL backend.

To summarize, the main contributions of this work are:

- An implementation of a compiler framework that uses vectorizing compiler algorithms for optimization of a restricted set of Python loop nests
- A FPGA backend that generates clusters of application-specific parallel processing elements (PEs)

2 Background

Data dependence gives constraints on the possible ordering of statements in a program. The order in which statements are executed can have a profound impact on performance, which further depends on the target platform. For example, consider the example of matrix-matrix multiply that is shown with three different orderings in Figure 2. In Nesting A, dependence theory tells us that the *k* loop must run sequentially because it reads and writes the same memory location ($C[i][j]$) in every iteration. The theory also tells us that the *i* and *j* loops carry no dependence. Therefore, they can be executed in parallel, allowing us to shift the loops inward, as is shown in Nesting B and C.

How we choose to execute these loops will vary for each platform. On a CPU with vector units, we may choose Nesting C. With this configuration, we can vectorize over the inner *j* loop and parallelize over the outer *i* loop.

We can even consider directly mapping these loops to custom parallel processing elements (PEs) on FPGAs. For this consideration, we take Nesting B and parallelize over the *i* loop. On other platforms, Nesting B may be disadvantageous due to synchronizing n times throughout the program. However, synchronization on the FPGA is very fast due to custom barrier networks.

Dependence sometimes also allows us to expose parallelism in inner loops by sequentially executing outer loops. In the code below, all three loops carry dependences. However, if the *i* loop is executed sequentially, then both the *j* and *k* loops may be done in any order, including in parallel. This is allowed because sequentially executing the *i* loop ensures that the left side of the statement will never point to the same memory location as the right side of the statement does.

```
for i in range(1,10):
  for j in range(1,10):
    for k in range(1,10):
      A[i+1][j+1][k+1] = A[i][j][k] + B
```

The previous example demonstrates a commonly used optimization strategy in which loops that carry dependences are pushed to the outermost position in order to expose parallelism in the inner loops. We use this general strategy when targeting FPGA parallel PE generation.

3 Compiler Implementation

Our compilation process begins with a dense loop nest specified in Python using NumPy arrays, as illustrated in Figure 3. Our front-end then generates an intermediate XML representation that is interpretable by our optimizing compiler. The optimizing compiler analyzes the loop nest using dependence and does source-level transformations such as loop reordering, blocking, and unrolling. Finally, separate backends generate application-specific FPGA multiprocessors, as well as more traditional targets such as CPUs with vector units, and OpenCL-programmable GPUs.

The Python front-end is based on the Copperhead [3] framework. Copperhead compiles a small data-parallel subset of Python to Nvidia CUDA for execution on the GPU. To support compilation on GPUs, Copperhead makes several restrictions required for compilation. It requires kernels be statically well-typed and enforce this restriction using a Hindley-Milner type system. Copperhead's type system has special importance in *Three Fingered Jack* because, while dynamic typing on GPU would have been cumbersome and limit the applicability of compilation techniques, fully supporting the run-time dispatch required for a dynamically typed language on the FPGA would be impractical, if not impossible. For the purposes of this paper, we further restrict the Copperhead type system to support only 32-bit NumPy data types (integers and single-precision floating point).

Our most significant modifications to Copperhead are the addition of for-loops to the grammar and removal of the data-parallel primitives. We only support for-loops with fixed bounds, no control-flow in the loop body, and affine array indexing functions. These restrictions simplify compiler construction and enable fast dependence checking heuristics. Copperhead was designed to show the applicability of map-reduce style functional programming on the GPU. In contrast, we assert focusing on iterative constructs using dependence analysis is a better approach for constructing FPGA accelerators. However, previous work has shown success with restricted map-reduce support on FPGAs [11].

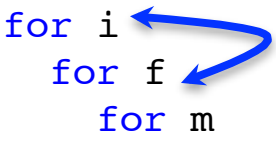
Figure 3 shows an example of the compilation process. When the Python interpreter encounters a function wrapped with the **@fpga** decorator, execution is redirected to our handler. As shown in subplot A of Figure 3, the **@fpga** decorator declares that the example kernel should be processed for FPGA compilation. After applying the syntax and semantic checking features of the Copperhead framework, we rewrite the GMM kernel abstract-syntax tree in XML so that we can export the kernel to our compiler. This process is shown in subplot B of Figure 3. We include the intermediate XML format so that our system will be able to support

```
@fpga
def GMM(In, Mean, Var, Out):
  for f in range(0,39):
    for i in range(0,5300):
      for m in range(0,16):
        Out[i][m] += Var[i][f][m] * \
           (In[f] – Mean[i][f][m])**2
```

(a) Original Python loop nest

```
...
<ForStmt>
  <LoopHeader>
    <Induction>f</Induction>
    <Start>0</Start>
    <Stop>39</Stop>
...
```

(b) XML intermediate representation



(c) Loop transformation

Fig. 3: Stages in the frontend.

other scripting languages in addition to Python.

The algorithms used to generate parallel PEs are similar to vectorization algorithms presented by Allen [1]. We focus on algorithms designed for vectorization instead of those for multiprocessor parallelization because vector instruction semantics are more desirable given our ultimate goal of hardware implementation. When a loop is vectorized, we guarantee it carries no dependence; therefore, each iteration of the loop proceeds in parallel. Our automatically generated processing engines operate in the spirit of vector processors. Instead of executing generic vector instructions (such as vector-add or vector-load), our PEs execute the entire body of loop as an application-specific vector instruction.

To enable vectorization, our compiler performs two key optimizations. First, it applies loop distribution to split the body of a multiple-statement loop into multiple smaller loops. Second, if a loop carries dependence, loop interchange shifts it to the outer-most legal loop position. This enhances vectorization opportunities. Subplot C of Figure 3 shows the application of loop interchange to a sample loop-nest. As shown, our compiler interchanges the *m* loop-nest to the outer-most loop in the example. After applying reordering transformations, our framework passes dependence analysis along with an intermediate representation to the FPGA backend.

PE cluster architecture As shown in Figure 4, the FPGA back-end generates PEs similar to vector-lanes from a traditional vector processor. As our PEs execute a single "virtual" vector instruction that potentially encompasses several dependence-free loops, we allow a limited amount of slip between PEs to tolerate memory latency effects. Slip refers to the case when PEs are running out of lockstep. Without a limited amount of slip, we would have to prevision significantly more memory bandwidth to tolerate multiple memory instructions occurring at the same cycle. We could have added a global stall network between PEs to handle memory request conflicts. Instead, we decided on PE cluster architecture that closely matches the structure of the regular code generated by our front-end.

In the dense kernels we use with our system, we have found that memory instructions occur approximately every 4 to 8 instructions. The current implementation of our LLVM to Verilog flow generates PEs with blocking memory operations, thereby simplifying the interaction with variable latency cached memory. Given that each PE has a blocking memory interface and executes a memory instruction every 4 to 8 instructions, a single PE cannot saturate our simple memory subsystem. Therefore, a cluster of PEs shares a global memory interface for maximal efficiency. We evaluate our automatically generated PEs with a relatively simple global memory subsystem. We use a 16 kByte, direct-mapped, write-back cache with 128-byte cache-lines to back a cluster of PEs.

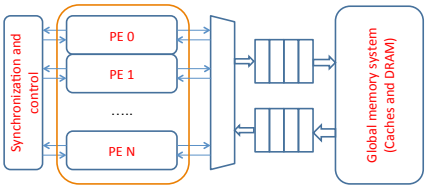


Fig. 4: FPGA processing cluster

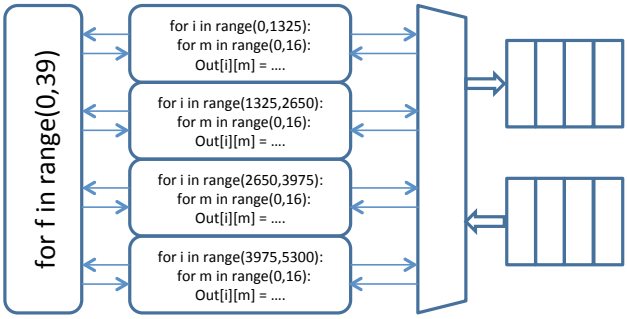


Fig. 5: GMM benchmark mapped onto FPGA PE template

To illustrate how computation occurs within a cluster of PEs, Figure 5 shows a mapping of a sample kernel. Note that the *f* loop is mapped to the synchronization and control component of the processing cluster and each of the 4 PEs executes a 1350-entry slice of the *i* iteration space.

FPGA back-end Our FPGA back-end uses the LLVM [7] framework because it enables straightforward code optimization and machine code generation passes. In addition, LLVM includes a vast repertoire of traditional compiler transforma-

tions that we apply to the intermediate representation generated by our vectorizing front-end. In particular, we apply dead-code elimination, loop-invariant code motion, and peephole optimization to optimize the IR generated by our front-end. We also exploit the ability to generate machine code for our x86 desktop CPUs in LLVM framework. We use this ability to check the correctness of our front-end.

To generate PEs, we map LLVM IR to Verilog RTL. Although our RTL generator is similar in principle to C-To-Verilog [12] and other systems that use LLVM for RTL generation, the architecture of the PE cluster requires a slightly different set of features than those provide. Above all, we need stalling memory support because contention for the shared memory interface and cache access introduces non-deterministic memory access times. We did consider manual modification of the RTL generated by an existing system, but this approach would have defeated our goal of automatically generating FPGA implementations from Python. We also briefly considered using an automatic clock gating scheme with C-To-Verilog to support stalling memory operations, but we decided it was simpler to implement our own tool.

Our RTL generation system uses conventional HLS algorithms [8].The user specifies the mix of functional units, latency, and support for pipelined operation. The system schedules the data-path using either list scheduling or an integer programming formulation [13]. A small library of operations supports single precision floating-point operations. Integer operations are generated behaviorally to support arbitrary pipelining depths.

4 Benchmarks

We evaluate our system on the following benchmarks. The benchmark sizes are implied in the loop-bounds. As our system is designed to handle regular dense loop nests, it is a natural fit for data parallel applications. We use kernels from complete applications instead of entire applications because we do not have access to a FPGA + multiprocessor SoC currently.

Vector-vector add (VV) A canonical data-parallel benchmark is the bandwidth-bound vector-vector addition routine:

```
for i in range(0,1024):
   c[i] = a[i] + b[i]
```

Color conversion (CC) We evaluate a simple color space conversion benchmark for a 128x128 pixel image. Color conversion can be expressed as a 3x3 matrix-transform applied to each pixel in an image. As the color conversion matrix is reused for each pixel, this benchmark has better memory reuse than vector-vector add does. Our implementation is shown below.

```
for p in range(0,16384):
 for i in range(0,3):
  for j in range(0,3):
   img_out[p][i] = img_out[p][i] +
                   img_in[p][j]*mat[i][j]
```

Matrix-matrix multiply (MM) Matrix-matrix multiply is a widely used kernel in dense linear algebra libraries. It serves as the workhorse for many higher-level algorithms, such as solving a system of linear equations, and it stresses the compute capability of most devices. Our matrix-matrix multiply is expressed as a triply nested loop:

```
for i in range(0,1024):
  for j in range(0,1024):
      for k in range(0,1024):
         c[i][j] += a[i][k]*b[k][j]
```

Gaussian mixture model evaluation (GMM) Modern speech recognition systems model the probability of a sound occurrence using a mixture of multivariate Gaussian distributions. It is common to use a mixture of 16 39-dimensional Gaussians per speech sound (phone). As noted in previous work [6], GMM evaluation accounts for greater than 50% of the run-time in the Sphinx3 system. The code used to evaluate a 5300 phone GMM is shown below:

```
for f in range(0,39):
   for i in range(0,5300):
      for m in range(0,16):
         LogProb[i][m] = LogProb[i][m] +
         (In[f] - Mean[i][f][m])*
         (In[f] - Mean[i][f][m])*
         (InvVar[i][f][m])
```

5 Results and analysis

FPGA statistics For evaluation of our PE generation system, we used a Xilinx XC6VLX240T-1FFG1156 FPGA for our study, as Xilinx Zynq SoCs evaluation boards are not currently available to us. We fully intend to use the Zynq with our system when an evaluation board becomes available. To implement our designs, we used Synopsys Synplify Premier F-2011.09-SP1 for synthesis and Xilinx ISE 13.4 for mapping and place-and-route.

LUTs	DSP48s	BRAMs	Max Freq
5570	3	5	91 MHz

Table 1*:* RISC-V Soft-core statistics (including memory subsystem)

We use an in-order 5-stage RISC-V processor [14] with a 4 kB instruction cache to compare with our automatically generated PEs. The relevant FPGA statistics of the RISC-V design are listed in Table 1. We used our compiler to generate optimized C implementations for the soft-core CPU. We compiled to the RISC-V ISA using GCC 4.4.0.

The FPGA LUT usage statistics of the automatically generated PEs for each kernel are shown in Table 2. The corresponding DSP48 usage and system frequency are shown in Table 3. Resource usage grows linearly with number of PEs for all kernels. We use the same memory subsystem

	VVADD	CC	MM	GMM
1 PE	3989	4057	5342	5666
2 PEs	4219	4772	7452	8178
3 PEs	4568	5474	9592	10657
4 PEs	4879	6115	11641	13538
5 PEs	5135	6824	13670	15758
6 PEs	4832	7560	15554	17967
7 PEs	5134	8414	18022	20743
8 PEs	5414	9134	19522	22743

Table 2*:* FPGA LUT Statistics (including memory subsystem)

	VVADD	CC	MM	GMM
Max Freq (MHz)	165	160	166	169
DSP48s per PE	0	3	3	3

Table 3*:* FPGA Freq and DSP48 Statistics

for both the soft-core CPU and the automatically generated PEs, so we can directly compare LUT utilization between the two implementations. The vector-vector add kernel uses fewer LUTs than the soft-core CPU does for all configurations as multiplication is not required for address computation with a one-dimensional array. In contrast, a single matrix-multiply or GMM PE with memory subsystem requires approximately the same number of LUTs as our soft-core CPU does. The color conversion kernel falls between the extremes, as 4 color conversion PEs require approximately the same number of LUTs as the soft-core does.

Performance We present results for single-cycle global memory (Figure 6) and global memory backed by a 16kB shared write-back cache with 128-byte cache lines. We evaluate the cache-based systems with 1-cycle (Figure 7) and 11-cycle (Figure 8) cache reload to show the impact of conflict misses and DRAM latency, respectively. We use 11-cycle reloads because we expect approximately 44 cycle DRAM latency [16] (assuming PEs run at 91 MHz and memory interface runs at 400 MHz). We compute speed-up by comparing the number of execution cycles on the soft-core to the number of execution cycles on the PE engine. This does not account for the different in obtainable frequency between the soft-core and the PEs.

As shown in Figure 6, our PEs are highly scalable with single-cycle memory. The vector-vector add kernel scales to nearly 7× soft-core performance with 8 PEs. In addition, both vector-vector add and matrix-multiply scale to slightly greater than 6× soft-core performance with 8 PES. The GMM kernel has the poorest scaling, limited to a maximum of approximately 3× soft-core performance. The GMM kernel is limited by shared cluster memory bandwidth for configurations greater than 4 PEs. The scalability results with single-cycle memories are encouraging because they confirm our decision to multiplex the global memory access port. In addition, while single-cycle memory access is infeasible for large memories, selectively partitioning data to take advantage of block RAMs on FPGAs appears favorable.

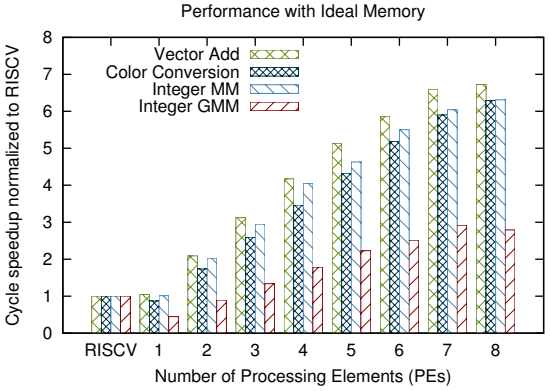


Fig. 6*:* Scaling with single-cycle global memory

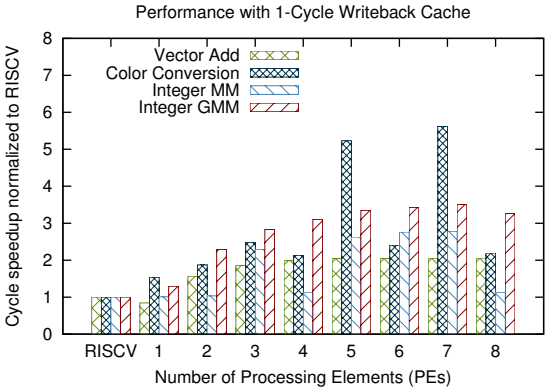


Fig. 7*:* Scaling with a writeback cache with 1-cycle reloads

Figures 7 and 8 show performance with the shared cache for 1-cycle and 11-cycle reloads, respectively. Maximum performance is obtained with 7 PEs due to cache conflict misses for both reload delays.

With 7 PEs and 1-cycle reloads, we obtain nearly 5× soft-core performance on the color conversion kernel. The other kernels achieve 2× and 3× performance increases with the same cache configuration.

The color conversion achieves greater than 5× soft-core performance with 11-cycle reloads due to reuse of the conversion matrix. The other kernels scale from 1× to nearly 4× with 11-cycle reloads. While scalability with cached memory is reduced compared with single-cycle memory; nevertheless, adding PEs increases performance. The addition of private caches to each PE would reduce conflict misses and improve performance. A LUT-efficient design would use less than 8 PEs as nearly peak performance is achieved with a smaller number of PEs.

6 Related Work

The theory of dependence and loop optimizations originated in the field of optimizing FORTRAN compilers for high-

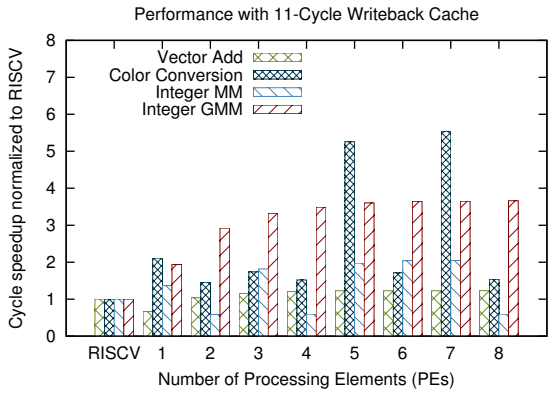


Fig. 8*:* Scaling with a writeback cache with 11-cycle reloads

performance computing in the 70s and 80s. Since then, several works have attempted to integrate these ideas into traditional high-level synthesis systems. Weinhardt [15] used dependence analysis on FPGA designs to enhance performance by executing independent iterations of a loop-nest through a heavily pipelined circuit. Dependence analysis enabled temporal multiplexing of a single data path to increase throughput. Work on the DEFACTO system [5] used dependence analysis in a similar fashion. In contrast, we use dependence analysis to generate parallel processing elements.

Recent work on *Irregular Code Energy Reducers* [2] (ICERs) provides a system for compiling existing irregular C code to specialized processing units. This tool uses a combination of compiler frameworks, including LLVM. There are two main differences between our work and the earlier work. First, we target dense loop nests and optimize for performance by generating parallel processing units, while the ICERs exclusively focus on energy and irregular serial code. Second, our high-level starting point is Python and our focus is programmer productivity.

FCUDA [9] is a HLS flow that maps the NVIDIA CUDA programming language to FPGAs. CUDA is an explicitly parallel language and requires the programmer to find parallelism. In contrast, we extract parallelism from a Python, a sequential language.

Our system is most similar to the non-programmable accelerators generated by the PICO system [10]. PICO uses dependence analysis to generate a systolic array of processing elements. Since the PICO system is proprietary, it is not available to the research community. Furthermore, we use Python as our source language instead of C.

7 Conclusions

We have evaluated the application of vectorizing transformations to automatically generate PEs from Python loop-nests and shown our approach is productive, portable, and efficient. Portability was guaranteed as all code remains valid Python while productivity was demonstrated by the brevity of our test kernels. Finally, efficiency was demonstrated by performance improvements from 3× to 6× relative to a soft-core CPU.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[2] M. Arora, J. Sampson, N. Goulding-Hotta, J. Babb, G. Venkatesh, M.B. Taylor, and S. Swanson. Reducing the energy cost of irregular code bases in soft processor systems. In *FCCM*, 2011.

[3] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *PPoPP*, 2011.

[4] B. Catanzaro, S. Kamil, Y. Lee, K. Asanovic, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. In *PMEA*, 2009.

[5] P. Diniz, M. Hall, J. Park, B. So, and H. Ziegler. Automatic mapping of c to fpgas with the defacto compilation and synthesis system. *Microprocessors and Microsystems*, 29(2-3):51–62, 2005.

[6] R. Iyer, S. Srinivasan, O. Tickoo, Zhen Fang, R. Illikkal, S. Zhang, V. Chadha, P.M. Stillwell, and Seung Eun Lee. Cogniserve: Heterogeneous server architecture for large-scale recognition. *Micro, IEEE*, 31(3):20 –31, may-june 2011.

[7] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *CGO*, 2004.

[8] A. McFarland, M. Parker and R. Camposano. Tutorial on high-level synthesis. In *DAC*, 1988.

[9] A. Papakonstantinou, K. Gururaj, J.A. Stratton, D. Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *SASP*, 2009.

[10] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B.R. Rau, D. Cronquist, and M. Sivaraman. Pico-npa. *Journal of VLSI Signal Processing*, 2002.

[11] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang. Fpmr. In *FPGA*. ACM, 2010.

[12] C to Verilog. Automating circuit design. `http://www.c-to-verilog.com`.

[13] R.A. Walker and S. Chaudhuri. Introduction to the scheduling problem. *Design Test of Computers, IEEE*, 12(2):60 –69, summer 1995.

[14] A. Waterman, Y. Lee, D. Patterson, and K. Asanović. The risc-v isa manual, volume i. Technical Report UCB/EECS-2011-62, May 2011.

[15] M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE TCAD*, pages 234 –248, 2001.

[16] Xilinx. Ug406: Virtex-6 fpga memory interface solutions.