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 accelerated on FPGA run 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.

We present *Three Fingered Jack*[1], a vectorizing compiler and high-level synthesis (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 run-time 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 work is inspired by the Selective Embedded Just-in-time Specialization (SEJITS) methodology, which uses embedded domain-specific languages to help mainstream programmers target Nvidia GPUs and multicore CPUs [3]. We extend the SEJITS ideas to target FPGAs. Additionally, our compiler is extensible and can support multiple backends. Though not the focus of this work, we also have a CPU backend and limited support for GPUs with an OpenCL backend.

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

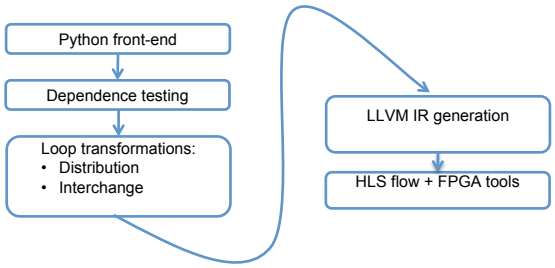

Fig. 1: Our compiler flow

```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
      Y[i][j] += A[i][k]*B[k][j]
```
Nesting A

```
for(k=0;k<n;k++)
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      Y[i][j] += A[i][k]*B[k][j]
```
Nesting B

```
for(i=0;i<n;i++)
  for(k=0;k<n;k++)
    for(j=0;j<n;j++)
      Y[i][j] += A[i][k]*B[k][j]
```
Nesting C

Fig. 2: Examples of matrix multiply loop interchange

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 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 ($Y[i][j]$) 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.

3 Compiler Implementation

Our compilation process begins with a dense loop nest specified in Python using NumPy arrays. 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, and code for CPUs with vector units and OpenCL programmable GPUs.

The Python front-end is based on the Copperhead [2] framework. Copperhead compiles a small data-parallel subset of Python to Nvidia CUDA for GPU execution. Copperhead requires kernels be statically well-typed. We further restrict the Copperhead type system to support only 32-bit NumPy data types in our system.

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.

When the Python interpreter encounters a function wrapped with the **@fpga** decorator, execution is redirected to our handler. After applying the syntax and semantic checking features of the Copperhead framework, we export the abstract-syntax tree as XML for our compiler.

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.

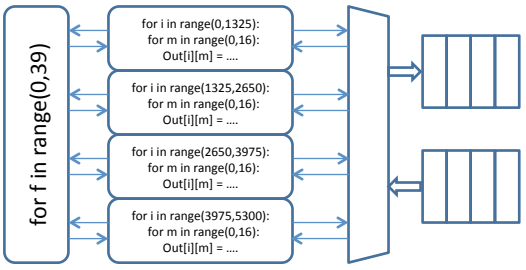

Fig. 3: GMM benchmark mapped onto FPGA PE template

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. After applying reordering transformations, our framework passes dependence analysis along with an intermediate representation to the FPGA backend.

PE cluster architecture 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.

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. Therefore, a single PE cannot saturate our simple memory subsystem. We exploit this by sharing the global memory interface within a cluster. We evaluate our PE clusters with a 16 kByte, direct-mapped, write-back cache with 128-byte cache-lines.

To illustrate how computation occurs within a cluster of PEs, Figure 3 shows a mapping of a benchmark 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 [5] framework because it enables straightforward code optimization and machine code generation passes. In addition, LLVM includes a vast repertoire of traditional compiler transformations 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.

To generate PEs, we map LLVM IR to Verilog RTL. Although our RTL generator is similar in principle other systems, our PE cluster requires additional features not supported by existing systems. Above all, we need stalling memory support due contention for the shared memory interface and non-deterministic memory access times due to caches.

Our RTL generation system uses conventional HLS algorithms [6].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. 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.

Vector-vector add (VV) Vector-vector add is the canonical data-parallel benchmark.

```
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.

```
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.

```
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.

```
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])
```

	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 1*:* FPGA LUT Statistics (including memory subsystem)

5 Results and analysis

FPGA statistics For evaluation of our PE generation system, we used a Xilinx XC6VLX240T-1FFG1156 FPGA for our study. To implement our designs, we used Synopsys Synplify Premier F-2011.09-SP1 for synthesis and Xilinx ISE 13.4 for physical implementation.

We use an in-order 5-stage RISC-V processor [8] with a 4 kB instruction cache to compare with our automatically generated PEs. We used our compiler to generate optimized C implementations for the soft-core CPU. When compiled to our Virtex-6 FPGA, the RISC-V soft-core requires 5570 LUTs, 3 DSP48s, and 5 BRAMs. It operates at 91 MHz. We compiled C benchmarks using GCC 4.4.0.

The FPGA LUT usage statistics of the automatically generated PEs for each kernel are shown in Table 1. The PEs run at approximately 160 MHz for all benchmarks. Resource usage grows linearly with number of PEs for all kernels. We use the same memory subsystem 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 4) and global memory backed by a 16kB shared write-back cache with 128-byte cache lines. We evaluate the cache-based systems with an 11-cycle (Figure5) cache reload penalty. We use 11-cycle reloads because we expect 44 cycle DRAM latency [10] (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 4, our PEs are highly scalable with single-cycle memory. The vector-vector add kernel scales

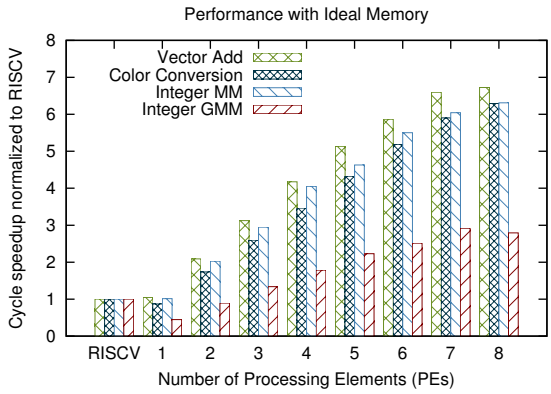

Fig. 4: Scaling with single-cycle global memory

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.

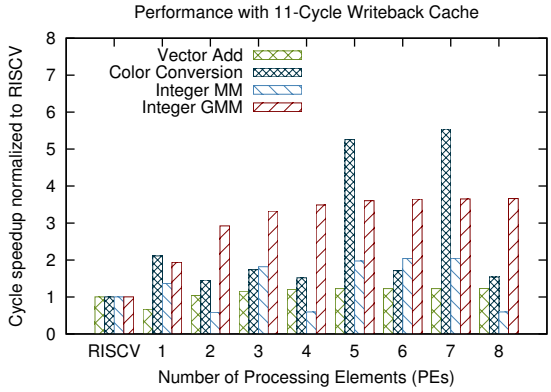

Fig. 5: Scaling with a writeback cache with 11-cycle reloads

Figure 5 shows shared cache performance with 11-cycle reloads. 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.

6 Related Work

The theory of dependence and loop optimizations originated in the field of FORTRAN compilers for numerical computing. Since then, several works have attempted to integrate these ideas into traditional high-level synthesis systems. Weinhardt [9] and Diniz [4] 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. In contrast, we use dependence analysis to generate parallel processing elements. Our system is most similar to the systolic array of PEs generated by the PICO system [7].

7 Conclusions

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

8 Acknowledgements

Research supported by Microsoft and Intel funding and by matching funding by U.C. Discovery. Additional support comes from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.

References

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

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

[3] 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.

[4] 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.

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

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

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

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

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

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