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

David Sheffield, Michael Anderson, Kurt Keutzer
{dsheffie,mjanders,keutzer}@eecs.berkeley.edu

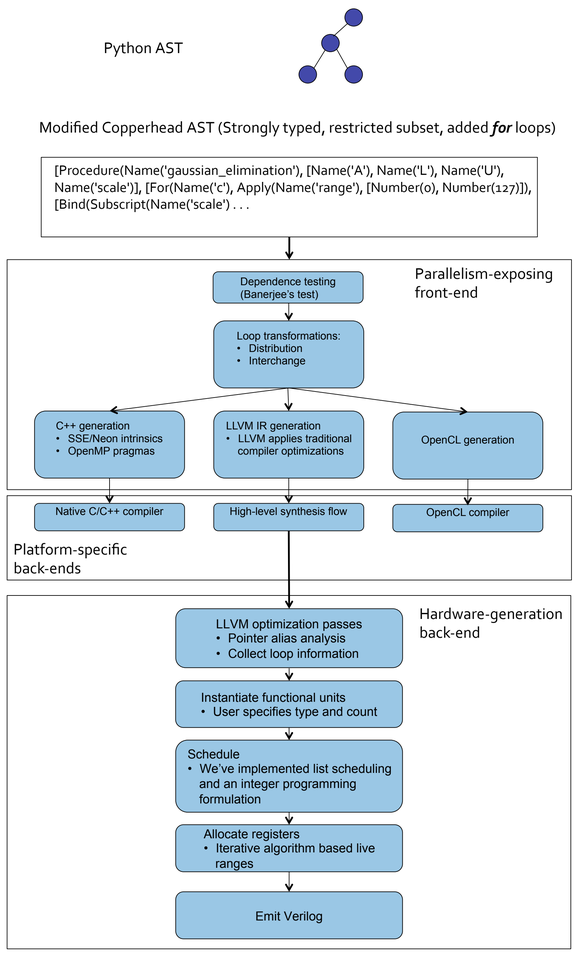
Introduction

- The emergence of SoCs with tightly coupled FPGA fabric and high-performance multicore CPUs encourages a new way of building FPGA-based systems
- FPGA+multiprocessor SoC allows the acceleration of select kernels on the FPGA fabric that closely match the capabilities of the fabric
- Parts of the program that can not be easily accelerated in the FPGA fabric can be run with reasonable efficiency on the multiprocessors
- *The FPGA+multiprocessor SoC motivates a selective and embedded to design: the programmer **selects** only certain computations for acceleration. These computations are **embedded** as a subset of a high-level language.*

Our approach

- We present *Three Fingered Jack* to productively explore application exploration on emerging FPGA+multiprocessor SoCs
- *Three Fingered Jack* is a vectorizing compiler and high-level synthesis system embedded in the Python language.
- In our system, the programmer selects dense loop nests in Python using the decorator syntax to redirect the Python run-time to our compiler
- As our compiler is restricted to dense loop nests, we can apply effective vectorizing compiler algorithms and traditional high-level synthesis techniques to automatically generate parallel processing engines
- Our approach has productivity, portability, and efficiency benefits.
  1. Portability is guaranteed as all code remains valid Python
  2. Efficiency is demonstrated by 3 to 6x performance improvements over an optimized soft-core processor
  3. We demonstrate productivity as our benchmark kernels are all less than 10 lines of Python. We believe high-performance manual RTL implementations would be at least 20x more code

Compiler Construction



Compiler Analysis and Transformations

- How do we unlock performance across a broad spectrum of hardware platforms…without manually coding implementations for each platform?
  - Reordering transformations!
- By changing the order of execution, we can better map computations to the underlying hardware

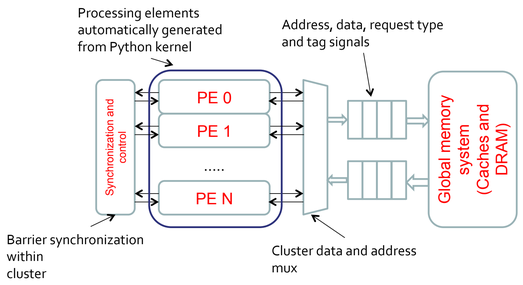
```
for(i=0;i<n;i++)
  for(j=0;j<n;j++)
    for(k=0;k<n;k++)
      Y[i][j] += A[i][k]*B[k][j]
```
Nesting A

```
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

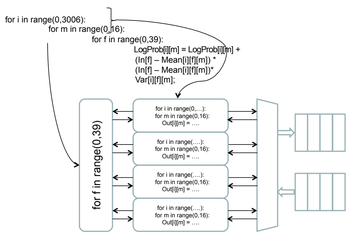
```
for(i=0;i<n;i++)
  for(k=0;k<n;k++)
    for(j=0;j<n;j++)
      Y[i][j] += A[i][k]*B[k][j]
```
Nesting C

- For example, code vectorization is enabled by moving a dependence-free loop to the inner most loop-nest
- We use Banerjee's test to construct our dependence graph
- Allen's codegen algorithm is at the heart of our compiler
- Aggressive optimizations: We perform loop distribution, loop interchange, and loop unrolling right now

Micro-architectural template



Micro-architectural mapping example



FPGA Evaluation and Results

We evaluated our system using the following Python kernels

- We used our system to generate both C and FPGA implementations of each kernel

Gaussian mixture model evaluation:

```
for i in range(0,3006):
 for m in range(0,16):
  for f in range(0,39):
   LogProb[i][m] += (ln[f] – Mean[i][f][m]) *
    (ln[f] – Mean[i][f][m])*
    Var[i][f][m];
```

Color conversion:

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

Vector add:

```
for i in range(0,1024):
 C[I] = A[I] + B[I];
```

Matrix multiply:

```
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]
```

Setup

- We evaluated our system on a Xilinx Virtex-6 LX240
- Synopsys Synplify Premier / ISE place and route
- Our Python high-level synthesis is built on Python 2.7 and LLVM 2.9
- Dependence testing and LLVM to RTL engines written for this project
- We compared the multiprocessors generated by our system to an optimized soft-core processor
- Scalar (five-stage pipeline) of the Berkeley RISC-V ISA
- C kernels compiled with GCC 4.4.0 with all optimizations enabled
- Automatically generated PEs run at 91 MHz
- Memory runs at 400 MHz

Design statistics

	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

Automatically-generated solution LUT statistics

LUTs	DSP48s	BRAMs	Max Freq
5570	3	5	91 MHz

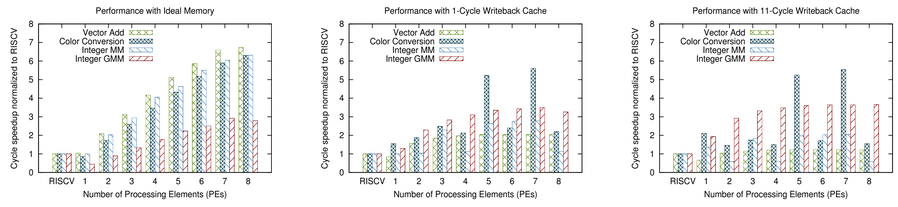
Soft-core statistics

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

Automatically-generated solution frequency statistics

Performance results

- Ideal memory: global memory accesses require a single cycle
- Cached results: main memory (DRAM) backed by 16kB direct-mapped write-back cache with 128-byte cache lines
- We use 1-cycle cache reloads to demonstrate the impact of conflict misses
- We use 11-cycle (PE clocks) cache reloads to demonstrate the impact of DRAM latency. 11-cycle reload due to 44-cycle DRAM reload (DRAM operates 400 MHz).



- Matrix-multiply, color-conversion, and vector-add are very scalable with single-cycle global memory accesses. DRAM bandwidth limits all GMM implementations
- With both 1-cycle and 11-cycle cache reloads, we obtain maximum performance with 7 PEs due to cache conflicts and limited memory bandwidth
- We achieve slightly less than 5x soft-core performance on the color-conversion kernel with 1-cycle reloads. The other kernels scale from 2x to 3x soft-core performance
- With 11-cycle reloads, we achieve greater than 5x soft-core performance with color-conversion kernel. The other kernels scale from 1x to 4x soft-core performance
- A LUT-efficient design built using our system would use less than 8 PEs. Adding private per PE caches would also improve performance significantly