Three Fingered Jack: Tackling Portability, Performance, and Productivity with Auto-Parallelized Python

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

UC Berkeley ParLab

HotPar’13, June 24, 2013
SIMD has arrived on the desktop and mobile

- Both mobile devices and desktop computers now include single-instruction multiple data (SIMD) units
  - Potential for $O(\text{SIMD width})$ speed-ups!
- Programming difficulties have led to relatively little adoption outside of low-level efficiency code
- We propose using a methodology of just-in-time specialization to automatically generate SIMD instructions from a high-level Python representation

Nvidia Tegra3
4 cores w/ 4 wide SIMD

Intel Ivy Bridge
4 cores w/ 8 wide SIMD
Three Fingered Jack: Example

```python
@tfj
def matmul(A,B,Y,n):
    for i in range(0,n):
        for j in range(0,n):
            for k in range(0,n):
                Y[i][j]=Y[i][j]+A[i][k]*B[k][j];
```

1.4 Mflops/s  
36 Gflops/s

- We start with matrix multiply written as three nested loops in Python
- By adding the `@tfj` decorator the Python runtime redirects execution to our framework
  - If we can not optimize the loop nest, it will be executed by the Python interpreter
Three Fingered Jack (TFJ) is a subset of Python used to generate GPU, CPU, and custom processing engine implementations.
Why target loops?

- Recent work in this space has focused on Map-Reduce data-parallel style programming frameworks [1,2]

- Python3 removed the reduce() builtin
  - “Removed reduce(). Use functools.reduce() if you really need it; however, 99 percent of the time an explicit for loop is more readable.” [3]

- We chose to build our framework using for-loops and extracting parallelism with compiler analysis techniques [4]

We removed Python constructs that are not amenable to optimization.

Python AST adapted from [http://docs.python.org/2/library/ast.html](http://docs.python.org/2/library/ast.html)
We started with a classic parallelization algorithm [1] and modified it to:

- Find unit-stride memory accesses to enable vectorization on desktop and mobile CPUs
- Reorder loops such that multithreaded execution will be profitable

TFJ: Implementation

- Entire compilation process happens at runtime – JIT parallelization / vectorization
- Analysis and code generator implemented in 13k lines of C++ and our code generator is written in 9k lines of Python
Evaluation – Setup

- Five variants of each kernel or application
  - Naïve Python
  - Python Libraries
  - TFJ
  - Untuned C++
  - Hand-tuned C++

- We evaluated TFJ on two different platforms
  - Desktop: Intel Core i7-2600
    - 4 cores, 8 threads
    - 8-wide SIMD (AVX)
    - 3.4 GHz
    - LLVM 3.1 MCJIT used for code generation
  - Mobile: Texas Instruments OMAP4460
    - 2 cores, 2 threads
    - 4-wide SIMD (NEON)
    - 1.2 GHz
    - GCC 4.7.3 used for code generation

- Our benchmarks use single-precision floating-point numbers
  - NEON only supports single-precision
Evaluation – Kernels

- Vector-vector addition with vectors of length 16M
  - Canonical data-parallel benchmark
  - Should achieve memory bandwidth-bound performance

- 2048x2048 matrix-matrix multiply
  - Common kernel in many scientific, engineering, and multimedia applications
  - An efficient implementation should be compute-bound

- Diagonal sparse matrix-vector multiply
  - Diagonally-dominant matrix generated from conjugate gradient solution of Horn-Schunck optical flow

- Back propagation weight adjustment
  - Key computation in training neural networks
  - We adopted our implementation from Rodinia [1]

Evaluation – Kernel Performance Results

- **EvaluaGon**
- **Kernel Performance Results**

<table>
<thead>
<tr>
<th>Task</th>
<th>OMAP4460</th>
<th>i7-2600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Vector Addition</td>
<td>81.8</td>
<td>1073.3</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>3268.6</td>
<td>112882.3</td>
</tr>
<tr>
<td>Diagonal Sparse Matrix Multiply</td>
<td>11.28823</td>
<td>35886.9</td>
</tr>
<tr>
<td>Back propagation weight adjustment</td>
<td>24578</td>
<td>18473</td>
</tr>
</tbody>
</table>

- **MFlops/sec**
- **Pure Python**
- **Untuned C++**
- **Hand-tuned C++**
- **Python Libraries**
- **TFJ**
Big idea: when resizing an image, remove the “boring parts” of an image

Algorithm [1] uses convolution and dynamic programming to iteratively remove “uninteresting” connected paths of pixels

Algorithm [1] uses convolution and dynamic programming to iteratively remove “uninteresting” connected paths of pixels

Algorithm [1] uses convolution and dynamic programming to iteratively remove “uninteresting” connected paths of pixels

Speech recognition has recently become a hot application on mobile devices.

We modified a conventional speech recognizer [1] (written in C++) to work with TFJ:
- We embedded Python in the recognizer and reimplemented the core inference engine in Python to demonstrate the power of TFJ.
- 85% of the C++ run-time spent in kernels accelerated by TFJ.
- We use 60 seconds of audio in evaluation.
  - Runtime less than 60 seconds implies real-time performance.

Speech recognition is close to real-time on mobile!
Evaluation – How well are we doing?

- TFJ achieves ~65% of ICC’s best matrix-multiply performance
  - Intel’s BLAS library obtains greater than 5x better performance for the same problem
- Perhaps compiler-based optimizations are limited to a certain performance level for the foreseeable future
  - However, the SEJITS approach is selective
    - It can compose well with other specializers and libraries

Intel ICC 13.0.1 on an Intel SandyBridge i7-2600K
Conclusions & Future Work

- We have demonstrated
  - A high-performance vectorizing and parallelizing JIT framework embedded in Python

<table>
<thead>
<tr>
<th></th>
<th>Untuned C++</th>
<th>Hand-tuned C++</th>
<th>Python libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-2600</td>
<td>3.14×</td>
<td>0.8×</td>
<td>1.1×</td>
</tr>
<tr>
<td>OMAP4460</td>
<td>1.8×</td>
<td>0.6×</td>
<td>0.9×</td>
</tr>
</tbody>
</table>

- Work in progress
  - Backends for more targets
    - OpenCL
    - UC Berkeley / MIT vector-thread processors
  - More support for irregular control flow
    - If-conversion
  - Integrate with existing SEJITS frameworks
    - ASP [1]
  - Build more applications using TFJ

THANK YOU
BACKUP
Evaluation – TFJ’s Overhead

Time (s)

Front-end | Compiler analysis and code-generation | Run-time code execution

OMAP4460 | i7-2600 | OMAP4460 | i7-2600 | OMAP4460 | i7-2600 | OMAP4460 | i7-2600
Vector Vector Addition | 0.008 | 0.005 | 0.014821 | 0.01 | 0.05 | 0.05 | 0.09 | 0.0001
Matrix Multiply | 1.27 | 0.009 | 1.75 | 0.04375 | 2.16 | 2.3 | 0.001 | 0.0001
Diagonal Sparse Matrix Vector Multiply | 0.19875 | 0.015 | 40.225 | 0.002417647 | 0.0001 | 0.00142578 | 0.001
Back propagation weight adjustment | 0.4375 | 0.024117647 | 0.024117647 | 0.024117647 | 0.024117647 | 0.024117647 | 0.024117647 | 0.024117647
Key Idea: Generate, compile, and execute high performance parallel code at runtime using code transformation, introspection, and other features of high-level languages.

In invisibly to the user.

Selective Embedded JIT Specialization (SEJITS)

Productivity app

- `gmm()`
- `fRead()`
- `matmul()`

SEJITS

AST conversion

Compiler analysis

Machine code generation

Interpreted

Underlying hardware

Compilation

Hardware information
Evaluation – How well are we doing?

- TFJ achieves 98% of ICC’s best matrix-multiply performance
  - Intel’s BLAS library obtains greater than 5x better performance for the same problem
- Perhaps compiler-based optimizations are limited to a certain performance level for the foreseeable future
  - However, SEJITS approach allows multiple approaches to parallel programming to cooperate in the same environment
    - Use the right tool for each programming problem

Intel ICC 12.0.4 on an Intel SandyBridge i7-2600K, N=2048, V=Vectorized, P=Vectorized+Multithreaded
Key kernels in ParLab apps are vectorizable

<table>
<thead>
<tr>
<th>Application</th>
<th>Kernel</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speech Processing</td>
<td>GMM evaluation</td>
<td>Evaluate likelihood of MFCC features using multiple 39-dimensional Gaussians</td>
</tr>
<tr>
<td></td>
<td>GMM training</td>
<td>Expectation-maximization algorithm used to train GMMs for multimedia apps</td>
</tr>
<tr>
<td></td>
<td>Neural network training</td>
<td>Neural networks are used in both multimedia and speech apps</td>
</tr>
<tr>
<td>Contour Detection</td>
<td>Generalized Eigensolver for solving normalized cuts</td>
<td>Contour detection uses a diagonal sparse matrix-vector multiply in the eigensolver</td>
</tr>
<tr>
<td></td>
<td>K-means clustering</td>
<td>Similar to pair-wise vector computation but with conditional updates</td>
</tr>
<tr>
<td>Object Recognition</td>
<td>Pair-wise vector computation (e.g. $X^2$ distance)</td>
<td>Comparing extracted features with the trained model</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>Linear (Preconditioned Conjugate Gradient) solver</td>
<td>Key kernels include matrix-vector multiply, vector-vector add, and vector scale</td>
</tr>
<tr>
<td>Image Feature Extraction</td>
<td>2d convolution</td>
<td>Used in SIFT for image blurring</td>
</tr>
<tr>
<td>Support Vector Machines</td>
<td>Linear SVM classification</td>
<td>The key kernel in classification with linear support vector machines is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>matrix-multiply</td>
</tr>
</tbody>
</table>
TFJ applied to matrix-multiply

- Parallelizing compilers need static loop bounds for high quality results
  ```c
  void mm_n(float **Y, float **A, float **B, int n) {
    int i,j,k;
    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];
  }
  ```
  ICC (n = 2048)
  Static bounds: 6600 mflops/sec
  Dynamic bounds: 275 mflops/sec

- However, using our SEJITS-style approach, we always know static loop bounds and can apply compiler analysis at run-time

TFJ run-times include time to generate code

For large matrices
TFJ is much faster than ICC
Outline

- Selected Embedded JIT Specialization (SEJITS) approach
- Gaussian Mixture Model & Applications
- Covariance Matrix Computation & Code Variants
- Specialization
- Results
Parallel processing is here

“This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.”

- The Berkeley View
## Evaluation – Seam Carving Overheads

![Chart showing evaluation results](chart.png)

<table>
<thead>
<tr>
<th></th>
<th>Time (s)</th>
<th>Front-end</th>
<th>Compiler analysis and code-generation</th>
<th>Run-time code execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMAP4460</td>
<td>tfj_grad2d</td>
<td>0.0131841</td>
<td>0.056824</td>
<td>1.34032</td>
</tr>
<tr>
<td>i7-2600</td>
<td>0.0014019</td>
<td>0.0133722</td>
<td>0.0132449</td>
<td>0.0133722</td>
</tr>
<tr>
<td>OMAP4460</td>
<td>zeroKernel</td>
<td>0.00784302</td>
<td>0.0482178</td>
<td>0.0482178</td>
</tr>
<tr>
<td>i7-2600</td>
<td>0.000244141</td>
<td>0.00809407</td>
<td>0.00809407</td>
<td>0.00809407</td>
</tr>
<tr>
<td>OMAP4460</td>
<td>tfj_conv2d</td>
<td>0.000244141</td>
<td>0.0482178</td>
<td>0.0482178</td>
</tr>
<tr>
<td>i7-2600</td>
<td>0.0152009</td>
<td>0.0583191</td>
<td>0.136609</td>
<td>0.136609</td>
</tr>
<tr>
<td>OMAP4460</td>
<td>dyn_prog</td>
<td>0.000101614</td>
<td>0.013536</td>
<td>0.661258</td>
</tr>
<tr>
<td>i7-2600</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- The chart illustrates the time overheads for different benchmarks across various platforms.
- The platforms include OMAP4460 and i7-2600.
- The benchmarks include tfj_grad2d, zeroKernel, tfj_conv2d, and dyn_prog.
### Evaluation – Optical Flow Overheads

<table>
<thead>
<tr>
<th>Function</th>
<th>Front-end</th>
<th>Compiler analysis and code-generation</th>
<th>Run-time code execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMAP4460</td>
<td>8.38681</td>
<td>0.0044148</td>
<td>23.7426</td>
</tr>
<tr>
<td>i7-2600</td>
<td>1.75903</td>
<td>0.0522759</td>
<td>0.039101</td>
</tr>
</tbody>
</table>

- Time (s) for various functions on OMAP4460 and i7-2600 processors.