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

David Sheffield, Michael Anderson, Kurt Keutzer
Parallel Computing Laboratory, Department of EECS
University of California, Berkeley
{*dsheffie,mjanders,keutzer*}*@eecs.berkeley.edu*

Abstract

We present *Three Fingered Jack*, a highly productive approach to writing high-performance software in Python. Our system applies traditional dependence analysis and reordering transformations to a restricted set of Python loop nests. It does this to uncover and exploit vector and thread parallelism. Then it generates native code for multiple platforms. On four benchmark kernels and two applications, our system achieves $0.4 - 13.5\times$ and $0.97 - 113.3\times$ of hand written but not highly optimized C++ performance on Intel and ARM CPUs, respectively.

1 Introduction

Within the last decade, the trajectory of mainstream computer architecture has radically departed from the homogeneous uniprocessor model. Issues such as wire scaling, design complexity, and energy consumption have limited the scalability of conventional designs and require new solutions. A wide range of parallel computer architectures and accelerators have been proposed to solve scalability and energy issues; however, while these systems solve architectural issues, they do little to help improve programmer productivity. Parallel systems of these varieties differ wildly in programming models and middleware, resulting in low developer productivity and challenging program portability.

Manually implementing high performance data-parallel platforms first requires the programmer to uncover parallelism that can be executed on his or her target platform. After completing this arduous task, the programmer must massage his or her implementation to fit the data and thread-level parallelism programming constructs on the chosen platform. In addition, source code portability between data-parallel processors is difficult due to differences in hardware intrinsics and often requires complete application rewrites for a new platform. Requiring the programmer to manually find and exploit parallelism severely limits developer productivity and application portability when developing high-performance software.

As developing parallel software on a new platform routinely takes days to weeks when doing initial development, alternative algorithmic approaches are rarely explored. Constructing parallel software in a low-level language such as C++ with threading libraries and data-parallel primitives is far too unproductive and error-prone as case studies show sequential C++ programming is 2 to $5\times$ less productive than languages such as Python [19, 14]. Moving between parallel platforms, for example from an ARM CPU to x86 CPU, often requires extensive re-engineering due to differences in the data-parallel primitives (Neon vs SSE). Finally, parallel programs are rarely performance portable due differences in data-parallel hardware, memory hierarchy, and core count. In summary, parallel computing needs tools that address the portability, performance, and programming challenges of modern systems.

The productivity gains made by languages such as Python has often come at an extreme performance cost. These interpreted languages are often orders of magnitude slower than a compiled efficiency language such as C or C++. As a result, there have been several projects working to accelerate productivity languages for non-expert programmers. Both Copperhead [7] and Parakeet [21] use data-parallel primitives as the basis to extract parallelism for GPUs from a subset of Python. In contrast, we use reordering algorithms to extract parallelism from a loop nest to generate efficient code for multicore CPUs with vector-units. We assert loops are easier to understand for the majority of programmers than functional programming primitives such as map and reduce. The removal of the reduce primitive from the core syntax of Python3 supports our assertion that for loops are more readable for most [24]. There have been several attempts to compile subsets of Python into sequential C [1] or LLVM [3]; however, these approaches are not sufficient as they do not automatically extract parallelism. NumPy

and SciPy provide data structures and routines for common mathematical operations; such as vector and matrix operations. NumPy provides an efficient implementation of multidimensional arrays to provide functionally similar to MATLAB. NumPy accelerates many common linear algebra operations by using highly optimized vendor BLAS libraries. SciPy uses NumPy's multidimensional arrays and optimized linear algebra operations to provide support for computational areas, such as numerical optimization and sparse linear algebra. Expressiveness is limited to the operators supported by NumPy or SciPy. Addressing the parallel programming challenge with libraries is efficient; however, it only works when high-performance libraries exist. In contrast, our approach enables a developer to obtain within $2\times$ hand-tuned parallel software within a matter of minutes using Python.

In this paper, we address the issues of portability, performance, and productivity using Three Fingered Jack (TFJ), an auto-parallelizing and vectorizing embedded domain-specific language (EDSL) for Python loop-nests. 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 loop-nests, we can apply aggressive parallelizing compiler algorithms to the loop-nests to automatically generate high-performance parallel implementations. Our work is inspired by the Selective Embedded Just-in-time Specialization (SEJITS) methodology that uses EDSLs to help mainstream programmers target Nvidia GPUs and multiprocessor CPUs [8]. We extend the SEJITS ideas with algorithms from parallelizing compilers to target loop-nests.

Portability is guaranteed, as all code is valid Python and can always be executed in the interpreter. If a loop-nest cannot be compiled for parallel execution (for example, a branch in the inner-loop), we can still compile a large subset of Python for scalar execution on the host CPU. Code that cannot be compiled for efficient execution remains valid Python and will be executed in the Python interpreter. The results in Section 3 exemplify the cross-platform *performance* benefits of our system across several kernels and two complete applications. Finally, programmer *productivity* is enhanced by our system as the user is allowed to write his or her application in Python and selectively accelerate specific computations.

TFJ currently supports full LLVM [16] JIT compilation on x86 multicores and ARM processors with Neon SIMD extensions. In the rest of the paper, we provide a detailed description of the TFJ system and provide results for both kernels and two full applications: content-based image resizing and speech recognition. Compared to hand written but not highly optimized C++ implementations, our TFJ results show performance improvements of $0.4-13.5\times$ on ARM CPUs and $0.97-113.3\times$ on Intel CPUs.

```
@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];
```

Figure 2: Matrix-Matrix Multiply written in Python for TFJ

2 Internals and Implementation

Our compilation process begins with a dense loop nest specified in Python using NumPy arrays, as illustrated in Figure 1. Our front-end then generates an intermediate XML representation of the abstract syntax tree (AST) that is interpretable by our optimizing compiler. Our compiler analyzes the loop nest using dependence analysis to enable other transformations such as loop reordering, blocking, and unrolling. Finally, separate backends generate LLVM IR for JIT execution on x86 as well as C++ with vector intrinsics. TFJ's run-time interfaces with the Python interpreter to execute the compiled code. TFJ is implemented in approximately 13000 lines of C++ and 9000 lines of Python.

2.1 Python Front-End

Figure 2 shows an example of matrix-multiply coded to use the TFJ EDSL. When the Python interpreter encounters a function wrapped with the **@tfj** decorator, execution is redirected to our front-end. The front-end requires kernels to be statically well-typed, and it enforces this restriction using a simple type system. TFJ requires that every expression has an associated NumPy data type. We support loop-nests with no control-flow and affine array indexing functions. These restrictions simplify compiler construction and enable fast dependence checking heuristics. Our run-time based compilation has the benefit of dynamic program information. When TFJ accelerated functions are executed, loop bounds are clearly defined and the dimensions of underlying NumPy arrays are known.

To avoid recompilation, we employ a compiled code cache. If a loop-nest does not use dynamic scoping, future recompilations will not be required. The TFJ front-end always attempts to use the code cache first; however, coding using dynamic scoping will require recompilation every time it is called for correct execution. While we have spent considerable effort making all stages of the compiler as efficient as possible, running the complete compilation pipeline requires upwards of 50 msec per

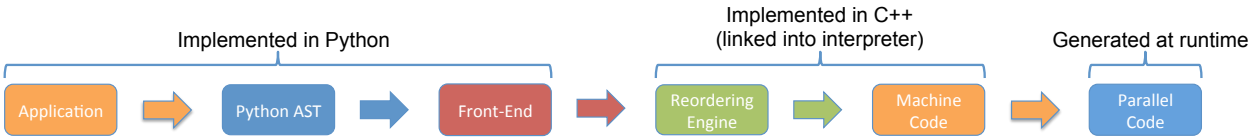

Figure 1: TFJ compiler flow: TFJ compiler flow: We start with computation expressed as a Python loop-nest. After syntactic and semantic checking in Python, we convert the Python AST to an XML representation that is fed to our reordering and optimization engine. The optimization engine then generates machine executable code for high-performance execution. Detailed descriptions of the front-end, reordering engine, code generator, and run-time are provided in Sections 2.1, 2.2, 2.3 and 2.4.

invocation on x86 and 3 sec on ARM. The long compilation times on our ARM-based PandaBoard-ES are due to slow IO performance of the Secure Digital (SD) card-based file-system.

We have a fall-back compiler that handles a larger subset of Python if and when TFJ rejects a loop-nest due to an unsupported construct. This EDSL does not attempt to exact any parallelism and therefore can handle a larger subset of the Python language, such as branches in loop-nests. We generate optimized sequential code using the same code generation framework described in Section 2.3. We do this because sequential Python code rejected by TFJ can become a performance bottleneck and thus in practice requires our fallback EDSL. We currently require the programmer to manually annotate code for the fallback compiler using a Python decorator.

2.2 Dependence Analysis and Reordering Transforms

Data dependence is the key to TFJ as it gives constraints on the possible ordering of statements in a program. Both the underlying hardware of the target platform and the order in which statements are executed can have a profound impact on performance. The benefits of dependence analysis and reordering transforms are best illustrated by an example. Figure 3 shows three legal orderings of matrix-multiply.

Dependence sometimes also allows us to expose parallelism in inner loops by sequentially executing outer loops. In Figure 3, all three loops carry dependences. However, if the i loop is sequentially executed, 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.

The inner-loop (K-loop) in Figure 3(a) carries a dependence which prevents inner-loop vectorization because the K loop must run sequentially. This is because it reads and writes the same memory location ($Y[i][j]$) in every iteration. However, dependence analysis also tells us that

```
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]
```
(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]
```
(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]
```
(c)

Figure 3: Three legal orderings of the matrix-matrix multiply loop-nest

the I and J loops carry no dependence and can be executed in parallel.

The nest shown in Figure 3(b) is amenable to inner-loop vectorization because the K-loop has been interchanged with the outer-most loop and it has unit-stride memory accesses. As our two conventional ISAs (x86 and ARM) include support for only unit-stride vector load and store instructions, finding an inner loop with unit-stride memory operations is required for vectorization. Now, the outer-loop in Figure 3(b) carries a parallelization-preventing dependence. The middle-loop (I-loop) may be parallelized across multiple CPUs; however, the profitability of this scheme is not guaranteed due to synchronization overhead. This is because the work granularity is relatively small. Finally, Figure 3(c) shows a loop-nest that allows for both outer-loop parallelization across cores and inner-loop parallelism across vector units.

Our reordering engine uses a simple heuristic when searching the legal reordering space. We sort dependence-free loops by the size of their iteration space (how many times the loop executes) and select the loop with the largest iteration space for the outer-loop. We desire the outer-most loop to have a large iteration space to avoid the overhead of frequent thread creation if the loop nest is executed with multiple threads. We then select a dependence-free inner loop with unit-stride or constant memory accesses. If we cannot find a reordering that satisfies these criteria, we attempt to expose parallelism in inner loops by allowing outer loops to execute sequentially. This heuristic works well in practice compared to exhaustive search, for the codes described

in this paper. This reordering scheme is valid because by running the outer-most dependence-carrying loop sequentially, we are free to reorder all inner-loops as we see fit.

To enhance opportunities for vectorization, we also perform loop distribution. By distributing loops, we can independently reorder statements in a loop-nest in order to find loop-nest permutations with unit stride memory access as required by our Neon and SSE vector extensions. If loop distribution does not unlock opportunities for vectorization, we apply loop fusion on distributed loops to reconstitute the original loop nest.

We use Banerjee's Inequality [6] for dependence testing and reordering algorithms similar to Allen's [2]. A restricted form of TFJ's dependence engine was previously used to generate parallel processing engines using hardware high-level synthesis techniques [22]. The dependence engine used for high-level hardware synthesis was far less aggressive when attempting to find unit-stride memory access as it was not required for custom hardware implementations.

2.3 Code Generation

After dependence analysis, we generate both C++ and LLVM IR. Our C++ code uses an abstract inline library of vector functions to map different vector ISAs such as ARM Neon, Intel SSE, or IBM AltiVec. The results from our source-to-source translation backend can be used outside our embedded Python environment or on systems that the LLVM JIT poorly supports.

We use LLVM's MCJIT to generate machine code at runtime as it does an excellent job of mapping high-level representations of vector operations in the LLVM IR to the appropriate vector instructions in a processor ISA. TFJ relies on LLVM implementations of scalar optimizations such as common-subexpression elimination, loop-invariant code motion, and strength reduction. Due to the relatively immature nature of MCJIT for ARM, TFJ emits C++ with intrinsics and generates shared objects with GCC at run-time. We generate unaligned vector memory operations because addresses are not guaranteed to be aligned. While unaligned vector loads and stores are potentially slower than their aligned equivalents on older x86 micro-architectures, new micro-architectures significantly close the performance gap. For example, on the Intel Nehalem micro-architecture unaligned vector operates are as fast as aligned operations when the memory operation does not span two cache lines [12]. The recently introduced ARM Cortex-A15 also includes improved unaligned memory accesses [15].

We also use the C++ generation backend to generate kernels for stand-alone applications. On platforms that do not support Python (or host-based compilation), such as a research operating system or an embedded microprocessor without an OS, we can use TFJ to generate high-performance kernels which are used in a standalone C or C++ application. We are currently using the offline capability to build a standalone computer-vision based music synthesizer on the Tessellation operating system [11].

Independent of the backend selected, both MCJIT and C++ generation of shared objects provide a function pointer to the JIT compiled code. We register both the function pointer and a hash of the Python source representation with the TFJ runtime system to enable code caching. If a TFJ accelerated function does not change after compilation, we can avoid compilation and executed cached machine code. We also store additional meta-data along with function pointer for the run-time environment. This meta-data encapsulates if the function is multi-threaded and information, such as the function pointer's prototype, needed to actually call the JIT-compiled code at run-time.

2.4 Run-time

To execute a function accelerated by TFJ, our run-time first queries the code cache to check if a compiled version exists. If it does not exist, or the hash changed due to program modifications, the code must be recompiled and the code cache updated. Once the desired code is found, execution meta-data and a function pointer to the compiled machine code are returned to the runtime.

To execute a TFJ accelerated function, TFJ queries the Python interpreter to first check if arguments are supported NumPy datatypes. If the function uses non-NumPy types, the function is executed by the Python interpreter instead of TFJ. After checking argument types, TFJ finds pointer references for the arguments and the dimensions (shapes) of any multidimensional arrays used. The arguments are stored in an auxiliary array according to the order specified by the execution meta-data. The process of collecting arguments also includes querying the Python interpreter for shape information needed for address calculation used to access elements within a multidimensional array. Arguments and shape data is used to check for pointer aliasing among arrays used as function arguments. If arguments alias the function must be executed in the Python interpreter. By dynamically querying shape information at function call-time, we avoid recompilation when a TFJ accelerated function is called with a different-sized array. For example, statically including shape information would prevent a function compiled for matrices of size 1024×1024 from working with a matrix of size 1023×1023.

To execute an TFJ accelerated function, the run-time checks if the function allows for multi-threaded execution. If the function is single-threaded, the Python interpreter executes the function using the function pointer and auxiliary argument array. If the code has been

compiled for multi-threaded execution, the TFJ run-time forks multiple threads for parallel execution using the pThreads API. The Python interpreter blocks until the parallel execution completes.

3 Evaluation

We evaluated TFJ with four kernels and two applications from the UC Berkeley ParLab [4]. The ParLab applications were chosen to represent compelling new uses of parallel hardware and drive our research in software and hardware systems. The results are shown in Figure 4 and present a comparison to untuned and optimized C++ implementations of the same computation. The untuned C++ implementations are not parallel (thread or data) but they were performed either by experienced programmer or taken from existing applications. We also compare TFJ with optimized Python libraries, if they exist for the given computation. The Python libraries we use for comparison are all implemented in C/C++ for efficiency. For completeness, we also present results when the loop-nests used by TFJ are executed in the Python interpreter. These implementations are represented in the headings of Figure 4 by "C++ (Untuned)", "C++ (Hand-tuned)", "Python Libraries", and "Pure Python", respectively.

We ran our benchmarks on both x86-based desktop and ARM-based mobile systems. We used a PandaBoard-ES with a dual-core 1.2 GHz Texas Instruments OMAP4460 SoC and 1GB of LPDDR2 RAM for our mobile system. Likewise, we used a 3.4Ghz Intel Core i7-2600k with 8GB of RAM for our desktop system. Both systems run Linux. We used LLVM 3.1 and GCC 4.7.3 for code generators on x86 and ARM, respectively.

3.1 Kernels

```
@tfj
def bpnn_adjust_weights(delta, ndelta, ly
    , nlyp1, w, oldw):
for j in range(0, ndelta):
  for k in range(0, nlyp1):
    w[k][j+1]+= \
      ((0.3*delta[j+1]*ly[k])+\
      (0.3*oldw[k][j+1]));
    oldw[k][j+1]= \
      ((0.3*delta[j+1]*ly[k])+\
      (0.3*oldw[k][j+1]));
```

Figure 5: Back propagation weight adjustment kernel from the Rodinia benchmarks rewritten in Python for TFJ acceleration

3.1.1 Vector-vector add

Vector-vector add is the canonical data-parallel benchmark. We used NumPy to compare against a Python library, while our optimized C++ implementation is manually vectorized.

3.1.2 Matrix multiply

This benchmark uses 2048×2048 single-precision matrices for matrix multiply. Our Python source is shown in Figure 2. We used ATLAS BLAS [25] and NumPy for our optimized C++ and Python library comparisons, respectively.

3.1.3 Diagonal sparse-matrix vector multiply

Optical flow relates the motion of objects between two video frames. It is a key computation in many computer vision algorithms and many optical flow algorithms address the problem using the conjugate gradient method to solve a system of linear equations. When optical flow is solved with conjugate gradient, a diagonal sparse-matrix vector multiply (SpMV) dominates the solver runtime. We used Sundaram's C++ implementations [23] and the SciPy Python library for comparison.

3.1.4 Back propagation weight adjustment

Back propagation weight adjustment (shown in Figure 5) is used as part of training neural networks. The C++ implementations are from Rodinia [9] and we used the OpenMP accelerated version of the kernel as the optimized implementation. We were unable to find a Python library for this computation.

3.2 Applications

Kernel performance results are not enough to fully demonstrate a programming system; therefore, we evaluated TFJ on two full applications representative of emerging workloads: content-aware image resizing [5] and speech recognition [10].

3.2.1 Content-Aware Image Resizing

(a) Original image (b) Retargeted image

Figure 6: Original image and retargeted image after removal of 750 vertical seams

Section discussed	Kernel/Application	Device	C++ (Un-tuned)	C++ (Hand-tuned)	Python Libraries	Pure Python	TFJ
3.1.1	Vector-Vector Add (*Mflops/sec*)	OMAP4460	981.4	**1146.3**	613.4	4.2	679.1
		i7-2600	12,879.5	**13,084.4**	11,407.7	24.0	12,469.9
3.1.2	Matrix Multiply (*Mflops/sec*)	OMAP4460	40.4	**3,268.6**	2,292.7	<0.2	546.7
		i7-2600	317.0	**112,882.3**	71,570.2	1.4	35,886.9
3.1.3	Diagonal SpMV (*Mflops/sec*)	OMAP4460	130.1	175.3	114.0	68.5	**336.2**
		i7-2600	2,457.8	3,052.6	1,847.3	1.5	**4,321.0**
3.1.4	Back propagation (*Mflops/sec*)	OMAP4460	100.7	**101.4**	N/A	0.07	41.2
		i7-2600	668.3	2,568.3	N/A	0.41	**3,147.5**
3.2.1	Seam Carving (*Runtime in sec*)	OMAP4460	205.4	**68.5**	18,873.1	>86,400.0	81.2
		i7-2600	20.4	**4.6**	2,238.6	34,646.8	7.1
3.2.2	Speech Recognition (*Runtime in sec*)	OMAP4460	78.9	**53.1**	N/A	>86,400.0	64.5
		i7-2600	5.4	**2.8**	N/A	19,775.9	3.8

Figure 4: Performance results for the 4 kernels and 2 applications. **Bold** numbers indicate best results. For the 4 kernels, larger Mflops/sec values indicate faster implementations. Application performance is reported in seconds; therefore, shorter runtimes reflect higher peformance. We have marked categories N/A if we could not find a Python library that implements a given benchmark. On our ARM platform, several benchmarks did not complete in under 24 hours when executed as Python loop-nests.

Content-aware seam carving, shown in Figure 6, resizes images by removing "boring regions". The seam carving algorithm computes an energy function to determine interesting regions, then computes a connected path of least-interest through the image.

Our implementation of seam carving is shown in figure 7. We first blur the image using the function **conv2d** to remove noise and then use the function **grad2d** to compute the gradient. The gradient extracts edges from an image. Regions with a small gradient have few edges and are unlikely to be interesting. We use a two-dimensional convolution kernel to compute the gradient. After computing the image gradient, we compute the minimum cost path through the image using function **compute_cost**. The minimum cost seam is computed by backtracking through the memorization table generated by **compute_cost**. The backtrack function is strictly sequential and we accelerate it using our serial DSL. To compare against Python libraries, we use SciPy and NumPy implementations of convolution and gradient calculation. The **conv2d**, **grad2d**, and **compute_cost** kernels have been manually vectorized and parallelized in our optimized C++ implementation.

3.2.2 Speech Recognition

Our speech recognizer is built around a hidden Markov model (HMM) inference engine with a beam search approximation [20]. As shown in Figure 8, the inference engine has two key phases: observation probability calculation using a Gaussian Mixture Model (GMM) and next-word search. The observation probability computation, shown in Figure 9, computes the probability of phonemes (elementary units of speech) in a 10 ms acoustic sample. The observation probability computation consumes 60% of the run-time in our C++ implementation. Two other kernels are used to infer words and sentences. These kernels are a search (graph traversal) computation that we accelerate with our serial DSL . In our C++ implementation, the search phase consumes 25% of the run-time. Our recognizer is built on top of a heavily modified version of the ICSI Parallex decoder.

The inference engine has been reimplemented in Python while the rest of the recognition application remains in C++. This approach allows us to demonstrate the power of TFJ without entirely reimplementing the speech recognizer in Python. We evaluate our recognizer using 60 seconds of audio from the 5000-word Wall Street Journal corpus [18] and compare our TFJ results with two C++ implementations of the speech recognizer: no parallelization and extensive manual parallelization. Our speech recognizer has a state-of-the-art 11.4% word error rate on both platforms.

3.3 Results and Summary

A complete tabulation of our results is presented in Figure 4 while Figure 10 presents our speed-ups. Our speed-up results show TFJ obtains similar performance to optimized C++ code on both platforms. While our TFJ results are within 60% to 80% of the performance of hand-tuned C++; however, the productivity benefits of our system are enormous. Hand-tuning our simple C++ kernels for correctness and performance took several hours while tuning the whole applications of Section 3.2 took several weeks. In contrast, TFJ results achieved correct, high-

```
@tfj
def conv2d(I,O,K,ydim,xdim):
  for y in range(3,ydim):
    for x in range(3,xdim):
      for yy in range(-2,3):
        for xx in range(-2,3):
          O[y][x]+=\
            K[2+yy][2+xx]*\
            I[y+yy][x+xx];
@tfj
def grad2d(I,O,K,ydim,xdim):
  for y in range(3,ydim):
    for x in range(3,xdim):
        O[y][x]=\
          (I[y][x-1]-I[y][x])*\
          (I[y][x-1]-I[y][x])+\
          (I[y-1][x]-I[y][x])*\
          (I[y-1][x]-I[y][x]);
@tfj
def compute_cost(Y,G,ydim,xdim):
  for i in range(5,ydim):
    for j in range(5,xdim):
      Y[i][j]=G[i][j]+\
       min(min(Y[i-1][j-1],Y[i-1][j]),\
       Y[i-1][j+1]);
```

Figure 7: A portion of the TFJ accelerated kernels used in seam carving.

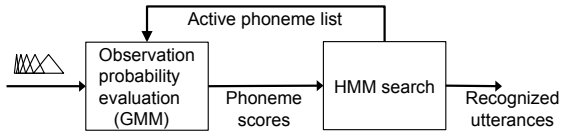

Figure 8: Architecture of our speech recognizer: 39-dimensional Mel-Frequency Cepstral Coefficients feature vectors are generated for each 10 ms acoustic sample. These feature vectors are fed to the inference engine to recognize words and sentences. The inference engine has two key phases: observation probability calculation using a Gaussian Mixture Model and a graph-based hidden Markov model search.

performance results within 15 minutes of developer effort for kernels and within hours for applications. For individuals requiring higher performance, our results are a good starting point for further code optimization as TFJ emits readable C++ than can be manually tuned.

We are particularly encouraged by our matrix-multiply results on the i7-2600 as our performance is within 33% of an optimized, auto-tuned library. In addition, the diagonal SpMV kernel produced by TFJ is over 3× faster than a call to SciPy's built-in diagonal SpMV routine on both platforms tested. Our speech recognition system using TFJ nearly achieves real-time performance on

```
@tfj
def GMM(In, Mean, Var, Out, Idx, n):
  for i in range(0,n):
    ii = Idx[i];
    for f in range(0,39):
      for m in range(0,16):
        Out[ii][m] += \
         (In[f]-Mean[ii][f][m])*\
         (In[f]-Mean[ii][f][m])*\
         (Var[ii][f][m]);
```

Figure 9: Observation probability evaluation kernel used in speech recognition written in Python for TFJ acceleration. Note that we use a linear map to access the Mean and Var arrays.

	Untuned C++	Hand-tuned C++	Python libraries
i7-2600	3.14×	0.8×	1.1×
OMAP4460	1.8×	0.6×	0.9×

Figure 10: TFJ speed-ups on Intel and ARM platforms compared to untuned C++, hand-tuned C++, and Python libraries.

the OMAP4460. We observe that the ARM Cortex A9 CPUs on our OMAP4460 SoC are more sensitive to manual code tuning, likely because the out-of-order engine is less aggressive than the design found in the Intel i7-2600. We are excited to experiment with the ARM Cortex-A15 when it becomes available for experimentation, as the both the out-of-order engine and vector unit has been significantly improved [15].

The two full applications demonstrate the power of the TFJ system as both applications are within 35% of the performance attained by manually tuned C++ across desktop and embedded CPU architectures. To further our portability goals, we are currently working on code generation for the vector-thread processors from UC Berkeley [17] and are currently tuning our system to generate efficient code for that platform. We are also working on a higher-level framework in Python for communication-avoiding matrix operations [13] that uses TFJ as one of its code-generation backends.

3.4 Acknowledgments

We thank Randy Allen for encouraging us to explore auto-parallelization within a SEJITS-based programming environment.

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates Nokia, NVIDIA, Oracle, and Samsung.

References

[1] Cython: C-extensions for python. 2010.

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

[3] Continuum Analytics. Numpy aware dynamic python compiler using llvm. `https://github.com/numba/numba`.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[5] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[6] U. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, November 1976.

[7] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 47–56, New York, NY, USA, 2011. ACM.

[8] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovic, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. Sejits: Getting productivity and performance with selective embedded jit specialization. In *First Workshop on Programmable Models for Emerging Architecture at the 18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[9] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44 –54, oct. 2009.

[10] Jike Chong, Ekaterina Gonina, Kisun You, and Kurt Keutzer. Exploring recognition network representations for efficient speech inference on highly parallel platforms. In *11th Annual Conference of the International Speech Communication Association (InterSpeech)*, pages 1489–1492, 2010.

[11] Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, David Zhu, John Shalf, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Resource management in the Tessellation manycore OS. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, Berkeley, CA, USA, June 2010.

[12] Martyn Corden. Compiling for nehalem. `http://ispass.org/ispass2010/tutorials/Compiling_for_Nehalem_Win_JR_DL.pdf`, 2008.

[13] Mark Frederick Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, Apr 2010.

[14] Paul Hudak and Mark P Jones. Haskell vs. ada vs. c++ vs awk vs..an experiment in software prototyping productivity. *Yale University Department of Computer Science Technical Report RR-1049*, 1994.

[15] Travis Lanier. Exploring the design of the cortex-a15 processor. `http://www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf`, 2011.

[16] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75 – 86, march 2004.

[17] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ISCA*, 2011.

[18] D.S. Pallett. A look at nist's benchmark asr tests: past, present, and future. In *Automatic Speech Recognition and Understanding, 2003. ASRU '03. 2003 IEEE Workshop on*, pages 483 – 488, nov.-3 dec. 2003.

[19] Lutz Prechelt. Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java. *Advances in Computers*, 57:205–270, 2003.

[20] M. Ravishankar. Parallel implementation of fast beam search for speaker-independent continuous speech recognition, 1993.

[21] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: a just-in-time parallel accelerator for python. In *Proceedings of the 4th USENIX conference on Hot Topics in Parallelism*, HotPar'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[22] D. Sheffield, M. Anderson, and K. Keutzer. Automatic generation of application-specific accelerators for fpgas from python loop nests. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 567 –570, aug. 2012.

[23] Narayanan Sundaram, Thomas Brox, and Kurt Keutzer. Dense point trajectories by gpu-accelerated large displacement optical flow. In *Proceedings of the 11th European conference on Computer vision: Part I*, ECCV'10, pages 438–451, Berlin, Heidelberg, 2010. Springer-Verlag.

[24] G. van Rossum. Whats new in python 3.0. `http://docs.python.org/3.0/whatsnew/3.0.html`.

[25] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.