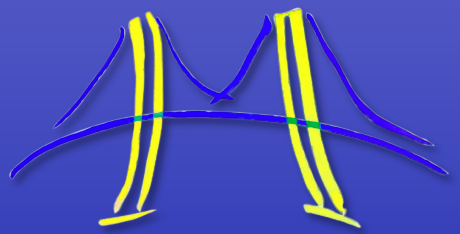
Three Fingered Jack: Tackling Portability, Performance, and Productivity with Auto-Parallelized Python

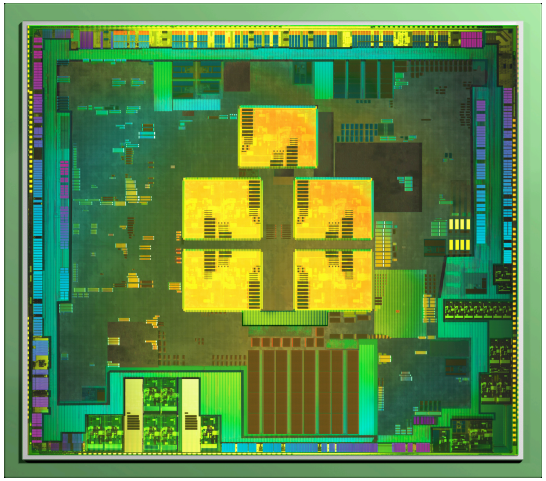
David Sheffield, Michael Anderson, Kurt Keutzer
{dsheffie,mjanders,keutzer}@eecs.berkeley.edu

UC Berkeley ParLab

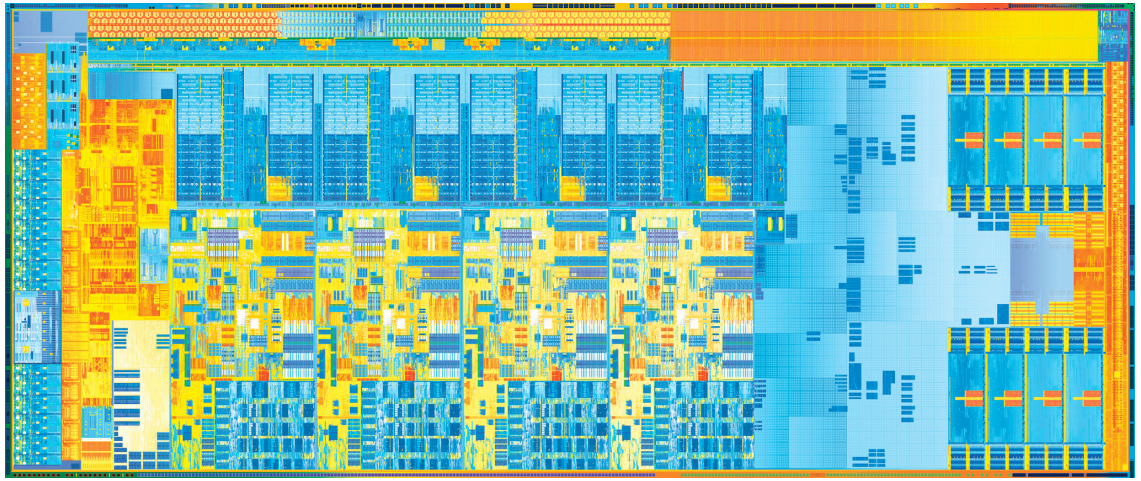
HotPar'13, June 24, 2013



SIMD has arrived on the desktop and mobile

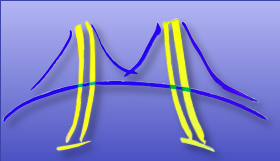


Nvidia Tegra3
4 cores w/ 4 wide SIMD



Intel Ivy Bridge
4 cores w/ 8 wide SIMD

- Both mobile devices and desktop computers now include single-instruction multiple data (SIMD) units
  - Potential for O(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



Three Fingered Jack: Example

```
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];
```

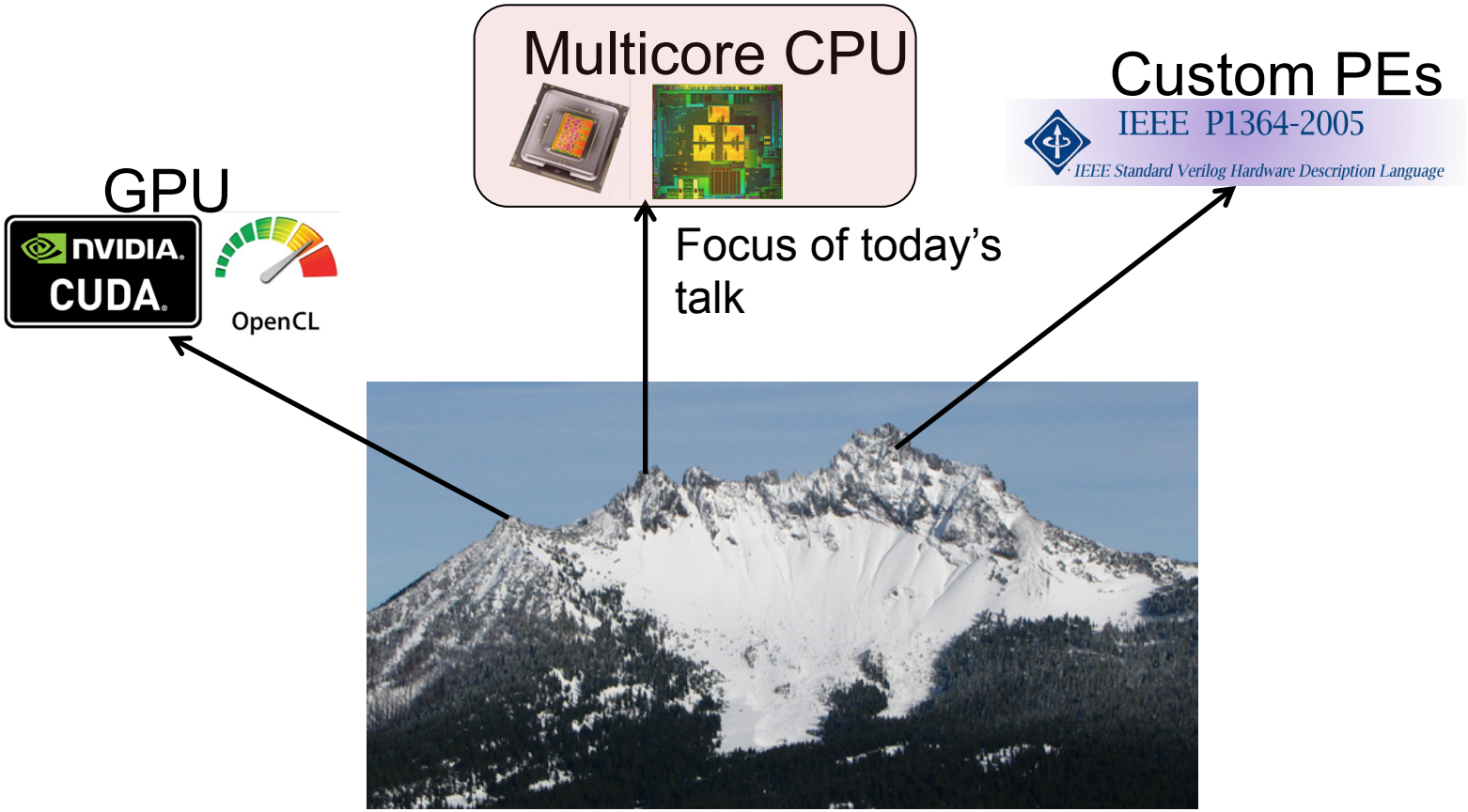

```
@tfj
def matmul(A,B,Y,n):
  for i in range(0,n):
    for j in range(0,n):
     for k in range(0,n):
       Y[i][j]=Y[i][j]+A[i][k]*B[k][j];
```

1.4 Mflops/s

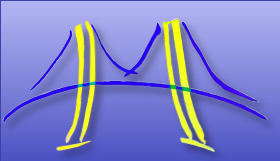
36 Gflops/s

- 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



- 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]

[1] Catanzaro, Bryan, Michael Garland, and Kurt Keutzer. "Copperhead: compiling an embedded data parallel language." *PPoPP*. ACM, 2011.
[2] Rubinsteyn, Alex, et al. "Parakeet: a just-in-time parallel accelerator for python." HotPar '12. USENIX Association, 2012.
[3] http://docs.python.org/3.0/whatsnew/3.0.html
[4] Allen, John R., and Ken Kennedy. "Automatic loop interchange." ACM SIGPLAN Notices. Vol. 19. No. 6. ACM, 1984.

Python as an optimization target?

```
mod = Module(stmt* body) | Interactive(stmt* body)
    | Expression(expr body) | Suite(stmt* body)

stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list)
     | ClassDef(identifier name, expr* bases, stmt* body,
                expr* decorator_list)
     | Return(expr? value) | Delete(expr* targets)
     | Assign(expr* targets, expr value)
     | AugAssign(expr target, operator op, expr value)
     | Print(expr? dest, expr* values, bool nl)
     | For(expr target, expr iter, stmt* body, stmt* orelse)
     | While(expr test, stmt* body, stmt* orelse)
     | If(expr test, stmt* body, stmt* orelse)
     | With(expr context_expr, expr? optional_vars, stmt* body)
     | Raise(expr? type, expr? inst, expr? tback)
     | TryExcept(stmt* body, excepthandler* handlers, stmt* orelse)
     | TryFinally(stmt* body, stmt* finalbody)
     | Assert(expr test, expr? msg) | Import(alias* names)
     | ImportFrom(identifier? module, alias* names, int? level)
     | Exec(expr body, expr? globals, expr? locals)
     | Global(identifier* names) | Expr(expr value)
     | Pass | Break | Continue
```

```
stmt = FunctionDef(identifier name, arguments args,
                   stmt* body, expr* decorator_list)
     | Assign(expr* targets, expr value)
     | For(expr target, expr iter, stmt* body)
     | Global(identifier* names)
     | Expr(expr value)

expr = BinOp(expr left, operator op, expr right)
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Num(object n)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)

expr_context = Load | Store
```

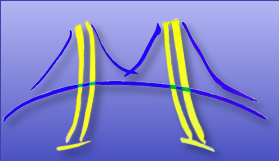
```
expr_context = Load | Store | Del | AugLoad | AugStore | Param

slice = Ellipsis | Slice(expr? lower, expr? upper, expr? step)
      | ExtSlice(slice* dims) | Index(expr value)

boolop = And | Or

operator = Add | Sub | Mult | Div | Mod | Pow | LShift
         | RShift | BitOr | BitXor | BitAnd | FloorDiv

unaryop = Invert | Not | UAdd | USub

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

comprehension = (expr target, expr iter, expr* ifs)

excepthandler = ExceptHandler(expr? type, expr? name, stmt* body)

arguments = (expr* args, identifier? vararg,
                         identifier? kwarg, expr* defaults)

keyword = (identifier arg, expr value)

alias = (identifier name, identifier? asname)
```

```
slice = Index(expr value)

operator = Add | Sub | Mult | Div | Max | Min

cmpop = Eq | NotEq | Lt | LtE | Gt | GtE

arguments = (expr* args
```

```
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts) | Yield(expr? value) | Repr(expr value)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords,
                       expr? starargs, expr? kwargs)
     | Num(object n) | Str(string s)
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)
```

- We removed Python constructs that are not amendable to optimization

Python AST adapted from http://docs.python.org/2/library/ast.html



Parallelization Approach

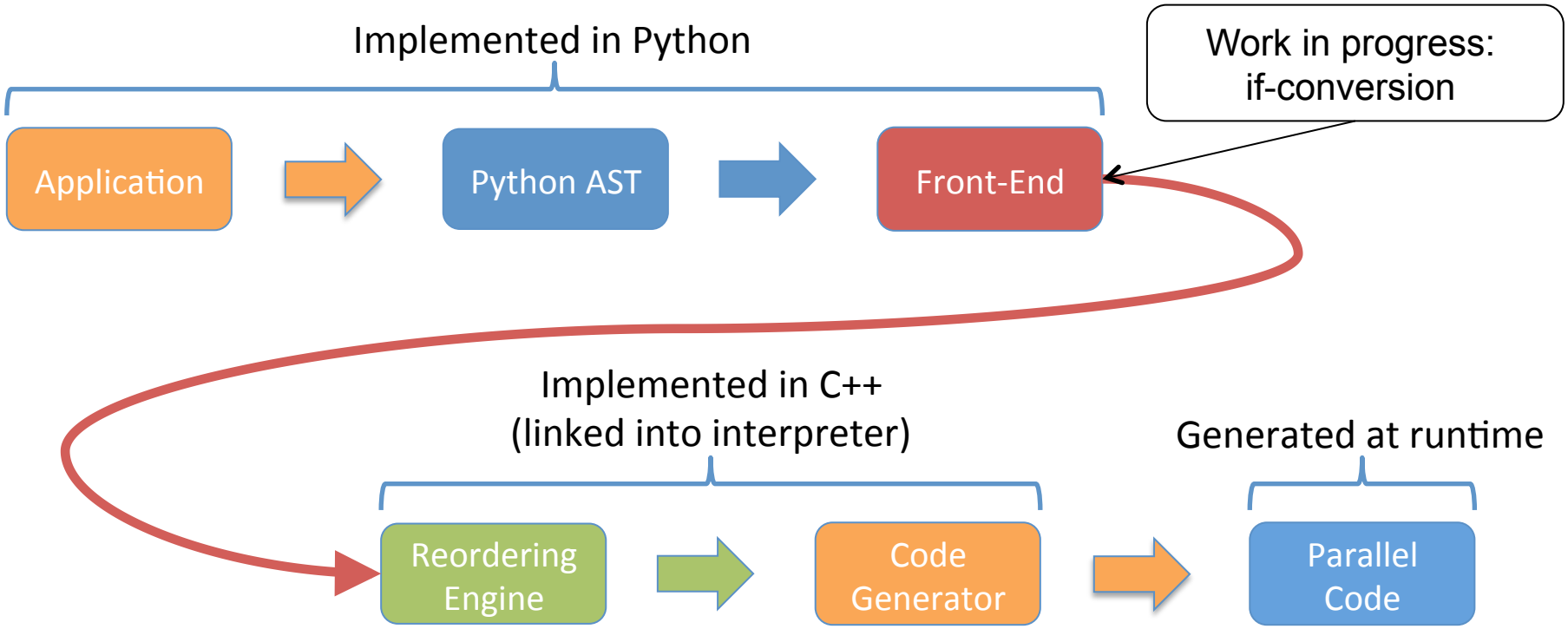
Algorithm: tfj_codegen
Input: R (Region to analyze for reordering)
k (Current loop nesting depth)
D (Dependence graph for the current region under analysis)
L (List of statements with associated dependence and nesting information)
Output: L (Updated list of statements)

1 Compute the set of strongly-connected components in
the dependence graph for the current region of interest
2 Topologically sort strongly-connected components according
to the dependence relationship to compute a set of $\pi_i \in \Pi$ blocks
3 **for** $\pi_i \in \Pi$
4 **if** π_i has a cycle (it can not parallelized at this loop-nest level)
5 If legal, attempt loop interchange by shifting dependence carried
at deeper loop nesting depth $(k+n)$ with the current level (k)
6 Remove new region graph, R_i by removing
all edges that are not in π_i
7 Update L_{in} with dependence and loop-nesting information for
the statements in π_i
8 Compute R_i and D_i with dependence and loop-nesting information for
the statements in π_i
9 Call **tfj_codegen**(π_i,$k+1$,D_i,L) for the region encapsulated by π_i
10 **else** (π_i can be parallelized at this loop-nest level)
11 Attempt to find a legal permutation of the loop-nests such that
a dependence-free loop is placed at the outer-most position and
a dependence-free loop with unit-stride memory access is placed
at the inner-most position, if found update nesting order for pi_i
12 Update L_{in} with dependence and loop-nesting information for
the statements in π_i

- 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

[1] Allen, Randy, and Ken Kennedy. Optimizing compilers for modern architectures. San Francisco: Morgan Kaufmann, 2002.

TFJ: Implementation

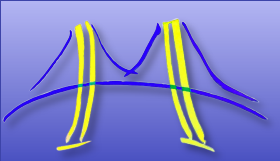


- Entire compilation process happens at runtime – JIT parallelization / vectorization
- Analysis and code generator implemented in 13k lines of C++ and our code generator is written in 9k lines of Python

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

PandaBoard ES

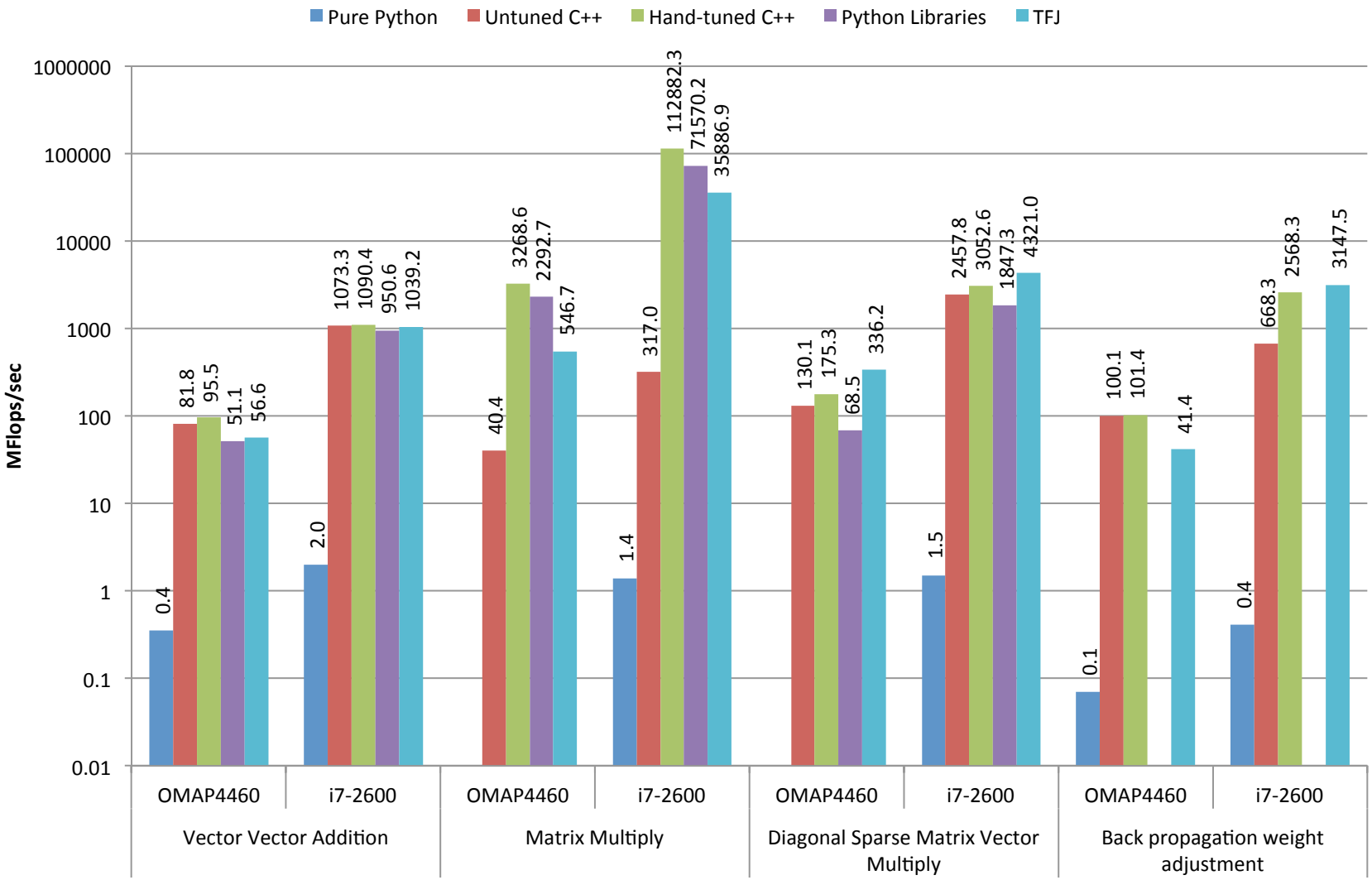


Evaluation – Kernels

- 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-dominate 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]

[1] Che, Shuai, et al. "Rodinia: A benchmark suite for heterogeneous computing." *IISWC 2009.* IEEE, 2009.

Evaluation – Kernel Performance Results



Evaluation – Content-Aware Image Resizing

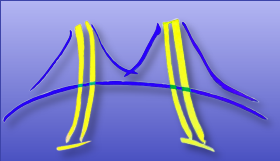
- Big idea: when resizing an image, remove the "boring parts" of an image

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

```
@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]);
```

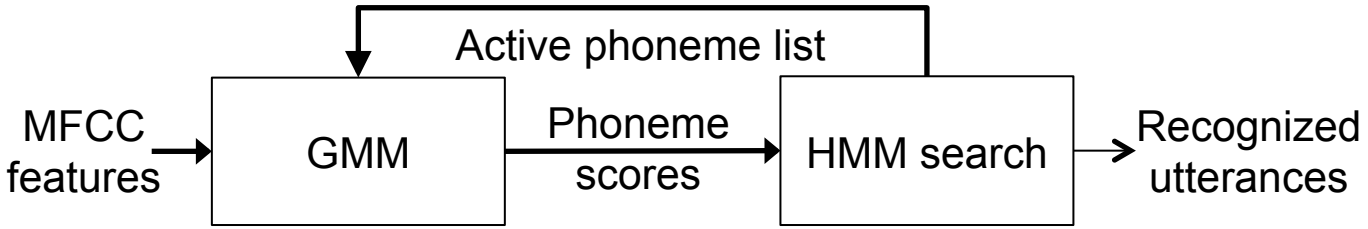
Actual kernels used in our Python implementation

[1] Avidan, Shai, and Ariel Shamir. "Seam carving for content-aware image resizing." *ACM Transactions on graphics (TOG)*. Vol. 26. No. 3. ACM, 2007.



Evaluation – Speech Recognition

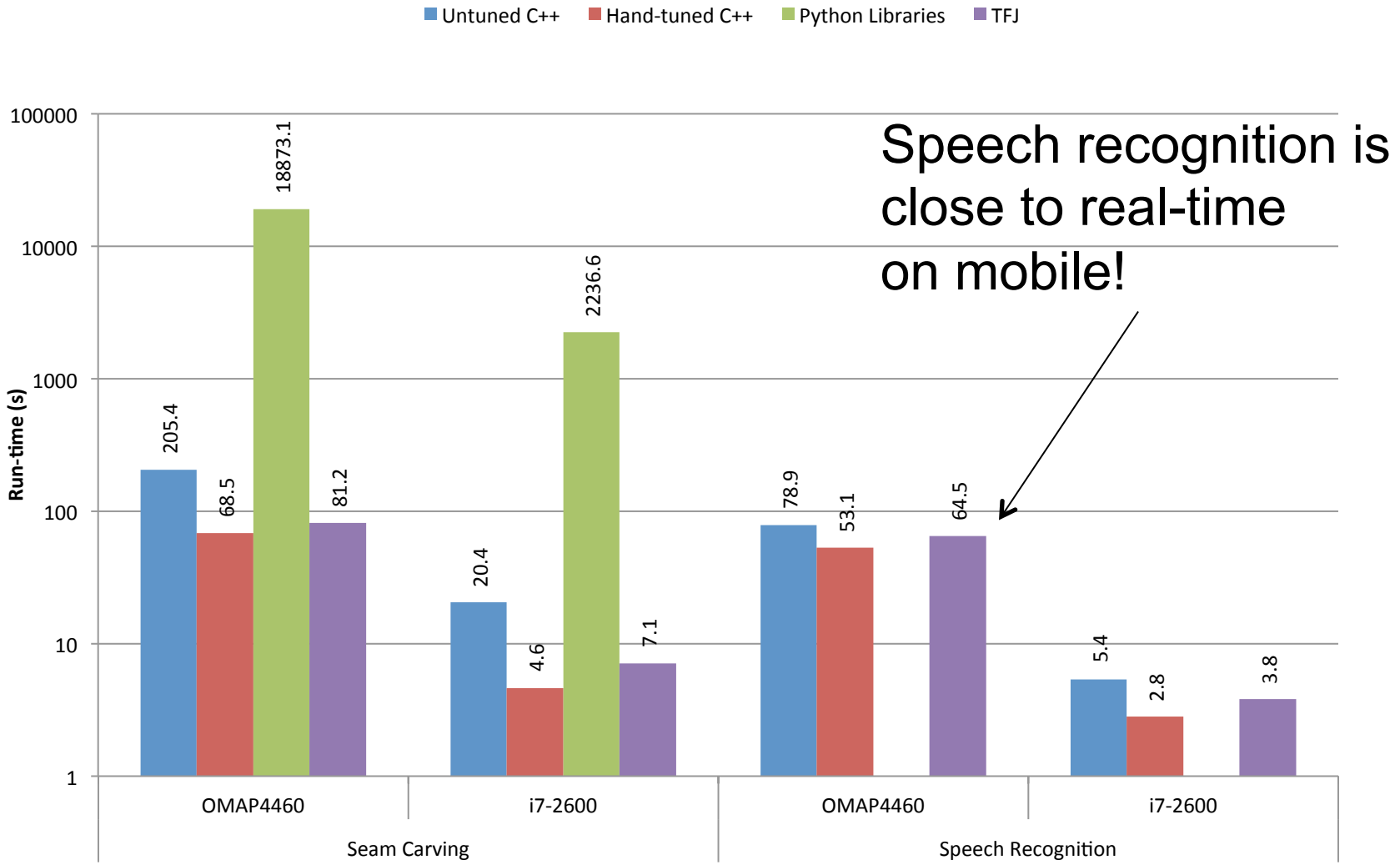
- Speech recognition has recently become a hot application on mobile devices



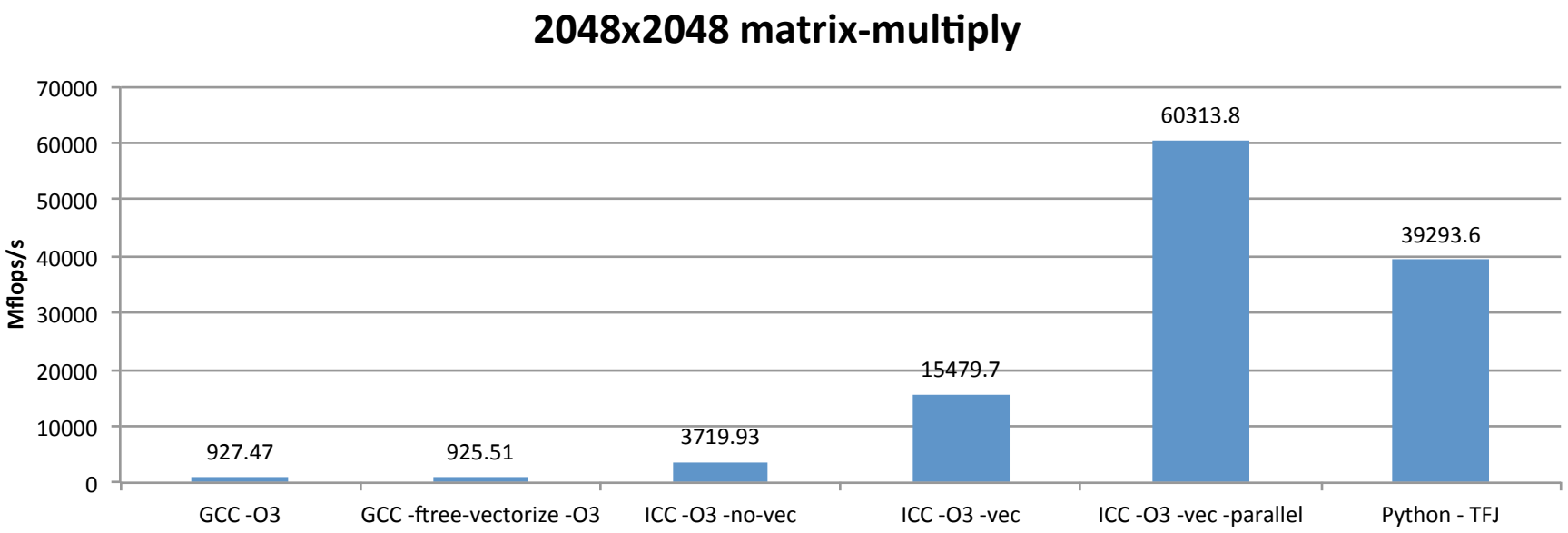
- We modified a conventional speech recognizer [1] (written in C++) to work with TFJ
  - We embedded Python in the recognizer and reimplemented the core inference engine in Python to demonstrate the power of TFJ
  - 85% of the C++ run-time spent in kernels accelerated by TFJ
- We use 60 seconds of audio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

[1] Chong, Jike, et al. "Exploring recognition network representations for efficient speech inference on highly parallel platforms." *InterSpeech*. 2010.

Evaluation – Application Performance Results

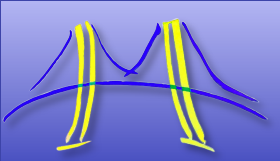


Evaluation – How well are we doing?



- 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

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

- 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

[1] Kamil, Shoaib, et al. "Portable parallel performance from sequential, productive, embedded domain-specific languages." PPoPP. ACM, 2012.

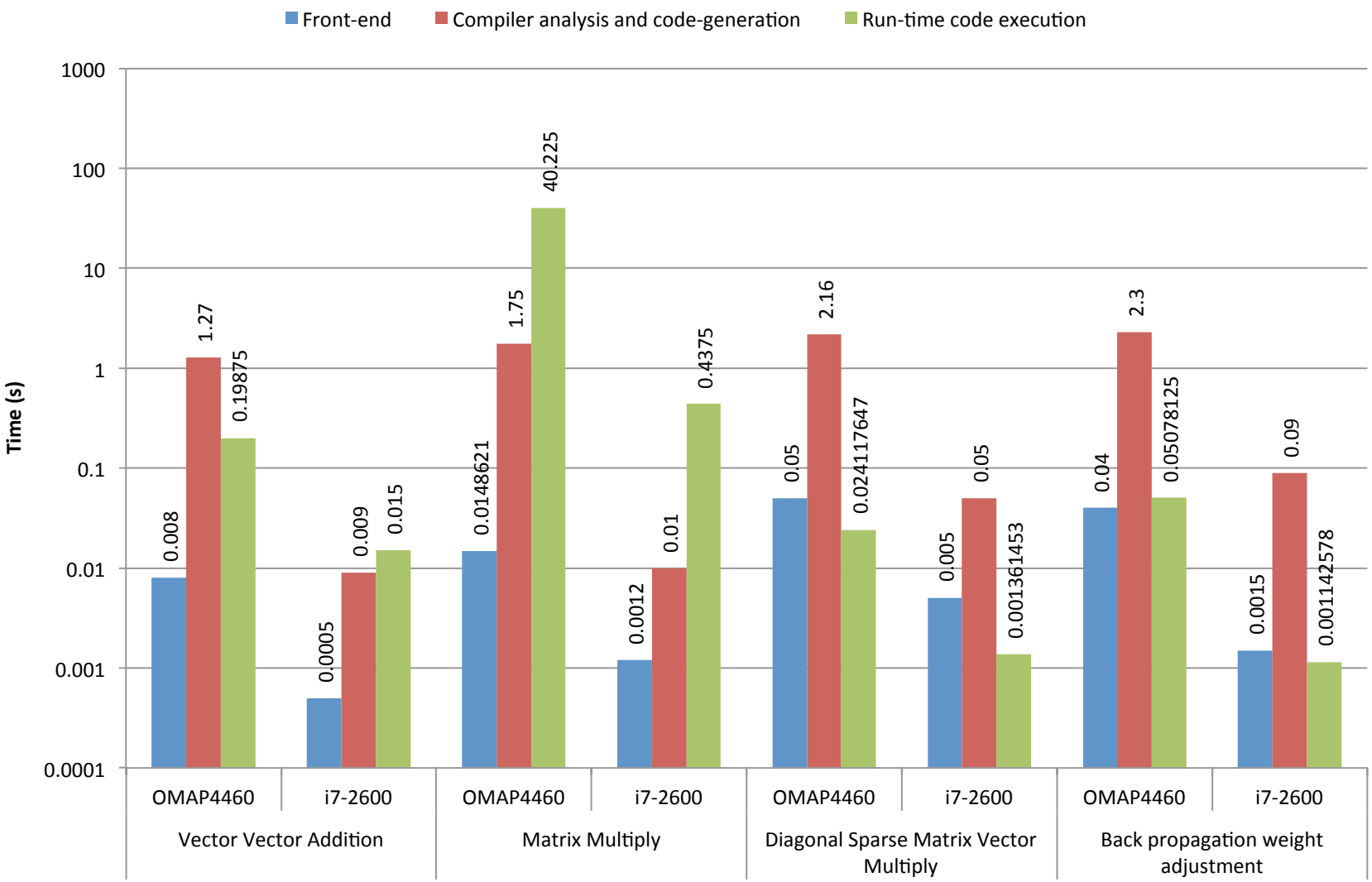
Questions?

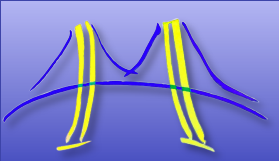
THANK YOU

Backup slides

BACKUP

Evaluation – TFJ's Overhead



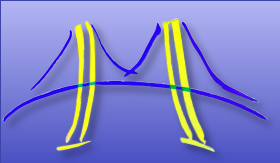


Selective Embedded Just-In-Time Specialization (SEJITS)

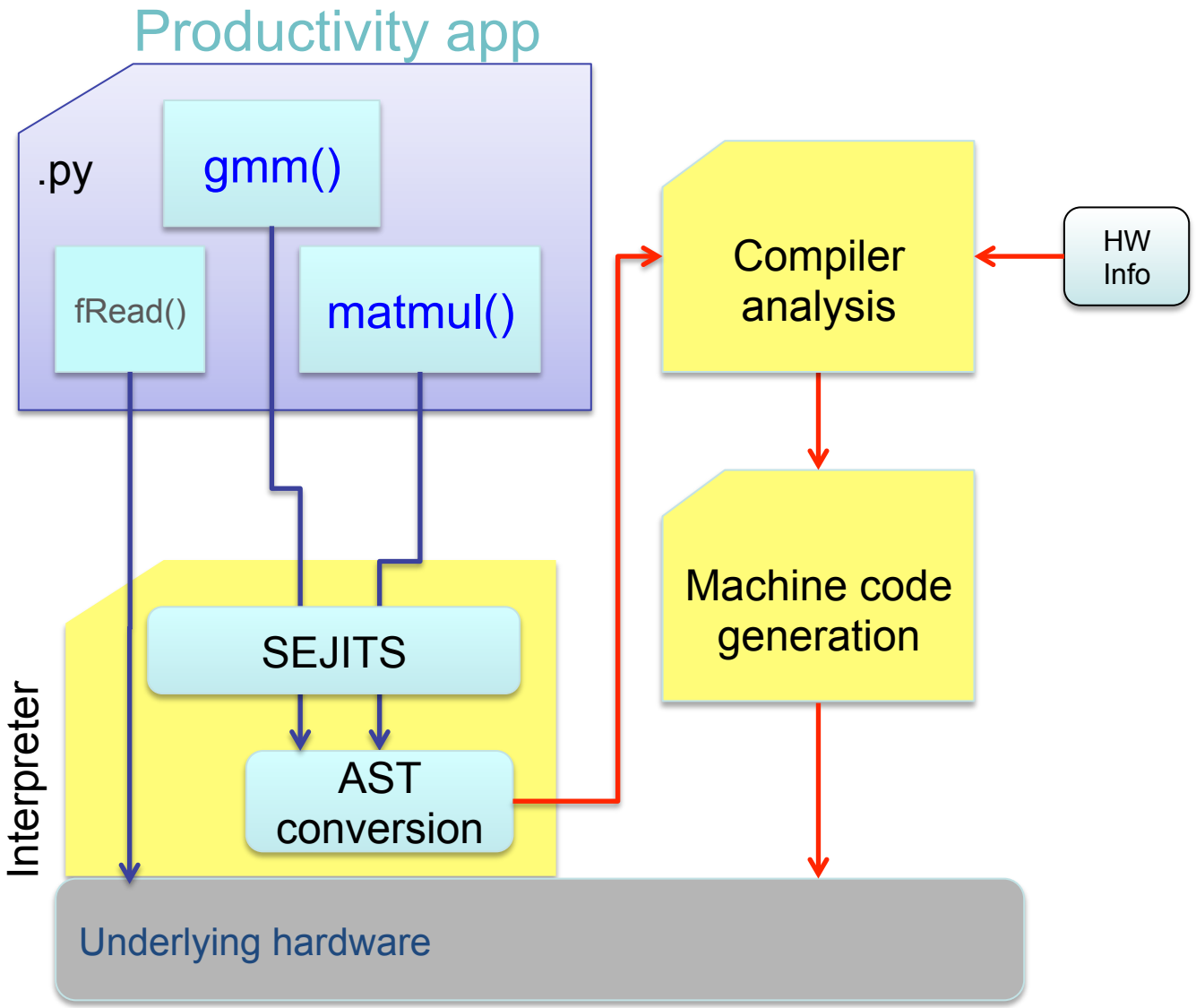
Key Idea: Generate, compile, and execute high performance parallel code at runtime using code transformation, introspection, and other features of high-level languages.

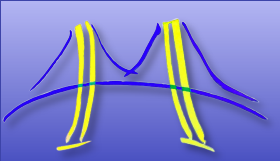
Invisibly to the user.

[1] 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 Workshop on Programming Models for Emerging Architectures (PMEA 2009), Raleigh, NC, October 2009.



Selective Embedded JIT Specialization (SEJITS)





Evaluation – How well are we doing?

```
void mm_v0(float **Y, float **A, float **B)
{
  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];
}
```

V: 2.2 GF/s
P: 6.6 GF/s

```
void mm_v1(float **__restrict__ Y,
           float ** __restrict__ A,
           float ** __restrict__ B)
{
  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];
}
```

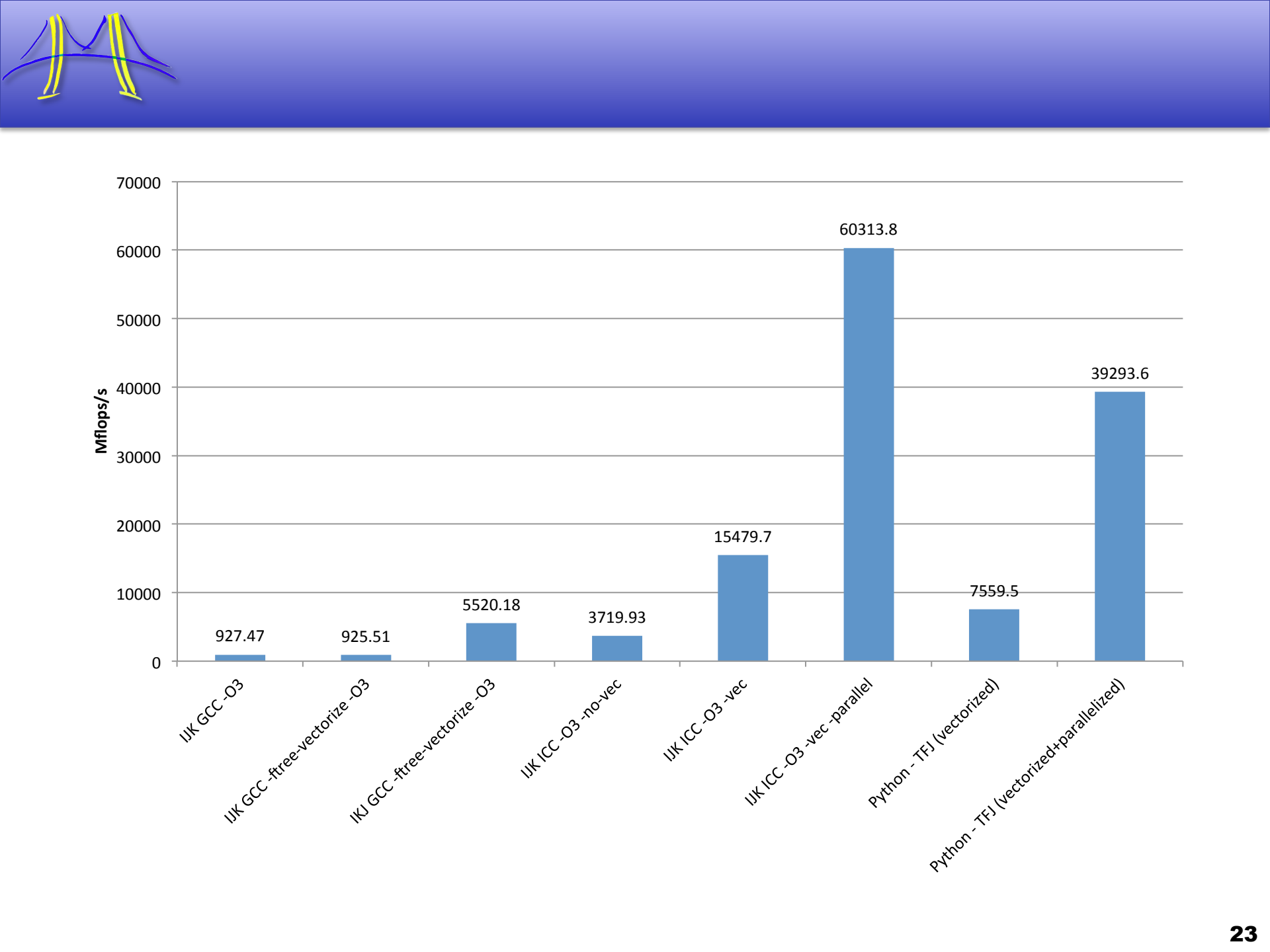
V: 2.2 GF/s
P: 6.6 GF/s

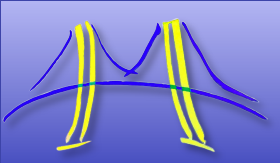
```
void mm_v2(float Y[N][N],
           float A[N][N],
           float B[N][N])
{
  int i,j,k;
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
      for(k=0;k<N;k++)
        Y[i][j] += A[i][k]*B[k][j];
}
```

V: 9.4 GF/s
P: 36.7 GF/s

- 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


Mflops/s
70000
60000
50000
40000
30000
20000
10000
0
927.47
925.51
5520.18
3719.93
15479.7
60313.8
7559.5
39293.6
IJK GCC -O3
IJK GCC -ftree-vectorize -O3
IKJ GCC -ftree-vectorize -O3
IJK ICC -O3 -no-vec
IJK ICC -O3 -vec
IJK ICC -O3 -vec -parallel
Python - TFJ (vectorized)
Python - TFJ (vectorized+parallelized)



Key kernels in ParLab apps are vectorizable

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

TFJ applied to matrix-multiply

- Parallelizing compilers need static loop bounds for high quality results

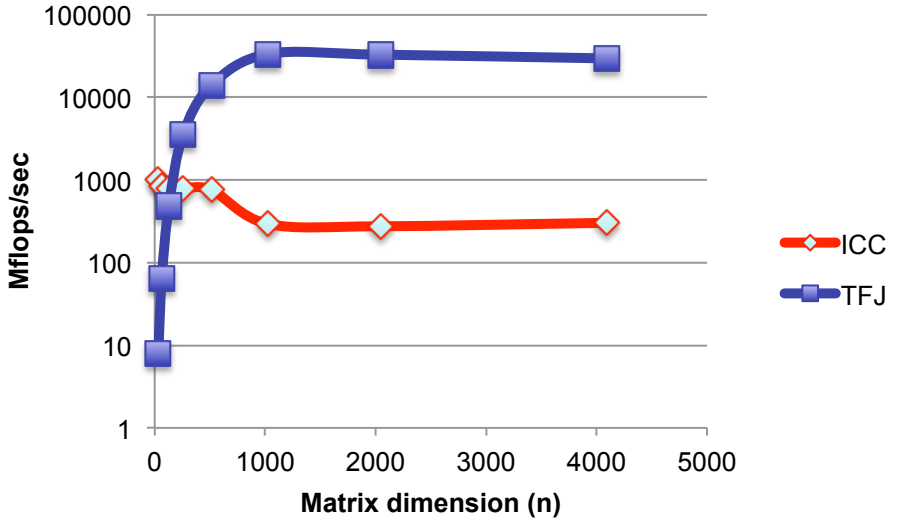
```
void mm_n(float **Y, float **A, float **B, int n) {
  int i,j,k;
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      for(k=0;k<n;k++)
        Y[i][j] += A[i][k]*B[k][j];
}
```

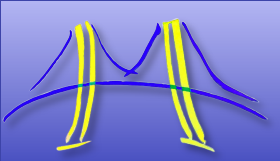
ICC (n = 2048)
Static bounds: 6600 mflops/sec
Dynamic bounds: 275 mflops/sec

- 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





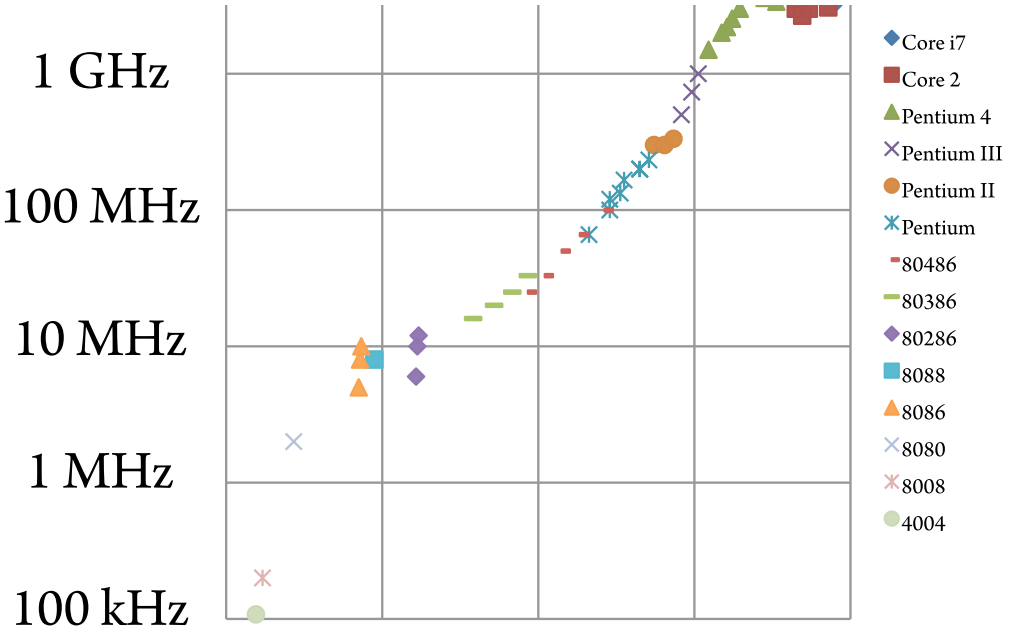
Outline

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

The shift to parallel processing

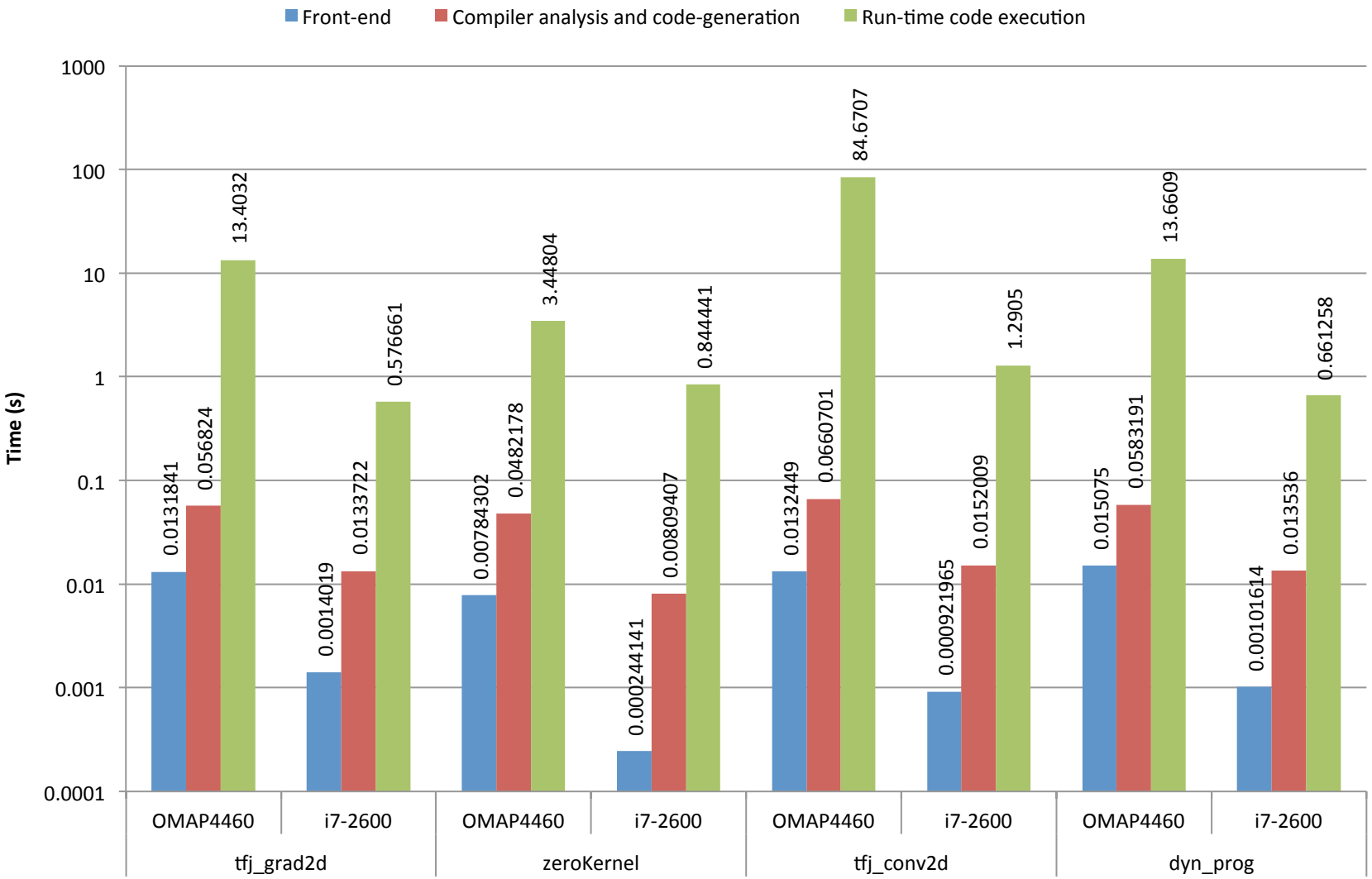
- Parallel processing is here

> *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



Intel Processor Clock Speed

Evaluation – Seam Carving Overheads



Evaluation –Optical Flow Overheads