Optimizing Matrix Mutliply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology

Jeff Bilmes, Krste Asanović, Chee-Whye Chin, Jim Demmel

{bilmes,krste,cheewhye,demmel}@cs.berkeley.edu

Computer Science Division, Dept. of EECS

International Computer Science Institute

U.C. Berkeley

Berkeley, CA 94707

Friday, July 11, 1997

11th ACM International Conference on Supercomputing

Presentation Outline

1. The PHiPAC Goal
2. Methodology
3. Fast Matrix-Multiply Code
4. Matrix-Multiply Benchmark Results
5. Future Work.

The PHiPAC Goal

Automatically and portably produce a high-performance matrix-vector library – maintain BLAS compatibility.

In general, ways to produce high performance code might include:

- Fixed assembly code
- High-level language: Fixed Code, Optimizing Compiler
- High-level language: Manually-tuned code, Optimizing Compiler

The PHiPAC Methodology

The PHiPAC strategy consists of 3 components:

1. Our C code, written according to special guidelines, is easy for a compiler to optimize – we rely only on good instruction selection, scheduling, and register allocation.
2. Rather than code a routine by hand either in assembly or a high-level language, we write parameterized C code generators.
3. We write scripts that search the parameter space to find the optimal parameters for a given compiler and microarchitecture.

PHiPAC Coding Conventions

The following summarizes the PHiPAC code generation guidelines:

1. Name registers as locals to remove false dependencies
2. Reduce memory bandwidth demands by using local variables
3. Use base + constant offset addressing mode to avoid unnecessary pointer updates
4. Expose independent operations to hide FPU latency
5. Attempt to balance 1 multiply, 1 add, and 1-2 memory ops
6. Exploit locality, structure for unit stride and data reuse
7. Use integer addition for multiply where possible
8. Minimize branch costs where possible
9. Manual loop unrolling and software pipelining to expose ILP

Name regs as locals removing false dependencies

Instead of:

```
a[i] = b[i]+c;
a[i+1] = b[i+1]*d;
```

we produce:

```
float f1,f2;
f1 = b[i]; f2 = b[i+1];
a[i] = f1 + c; a[i+1] = f2*d;
```

Reduce memory bandwidth demands via local vars

Instead of:

```
while (...) {
   *res++ = filter[0]*signal[0] +
            filter[1]*signal[1] +
            filter[2]*signal[2];
   signal++;  }
```

we produce:

```
float f0,f1,f2;
f0=filter[0];f1=filter[1];f2=filter[2];
while ( ... ) {
  *res++ = f0*signal[0]
         + f1*signal[1] + f2*signal[2];
  signal++;   }
```

Use base + constant offset addressing mode

Instead of:

```
f0=*r8; r8+=4;
f1=*r8; r8+=4;
f2=*r8; r8+=4;
```

we produce:

```
f0 = r8[0]; f1 = r8[4]; f2 = r8[8]; r8 += 12;
```

Integer addition for mult where possible

Instead of:

```
for (i=...) {row_ptr=&p[i*col_stride];...}
```

we produce:

```
for (i=...) {...row_ptr+=col_stride;}
```

Minimize branch costs wherever possible

Instead of:

```
for (i=0,a=start_ptr;i<ARRAY_SIZE;i++,a++}
    { ... }
```

we produce:

```
end_ptr = &a[ARRAY_SIZE]; a = start_ptr;
do { ... a++; } while (a != end_ptr);
```

Expose ILP via loop unrolling/software pipelining

For example:

```
float f0,f1,f2,s0,s1,s2;
f0=filter[0];f1=filter[1];f2=filter[2];
s0=signal[0];s1=signal[1];s2=signal[2];
*res++=f0*s0+f1*s1+f2*s2;
do {
   signal+=3;
   s0=signal[0];res[0]=f0*s1+f1*s2+f2*s0;
   s1=signal[1];res[1]=f0*s2+f1*s0+f2*s1;
   s2=signal[2];res[2]=f0*s0+f1*s1+f2*s2;
   res += 3;
} while ( ... );
```

Code Generators

A *Generator* is a program that, given a set of parameters, outputs C code according to our guidelines. Our matrix multiply generator, for example, is called as follows:

- `mm_gen -cb 2 6 2 -cb 24 10 4`

Our generators have several advantages over hand or assembly coded routines:

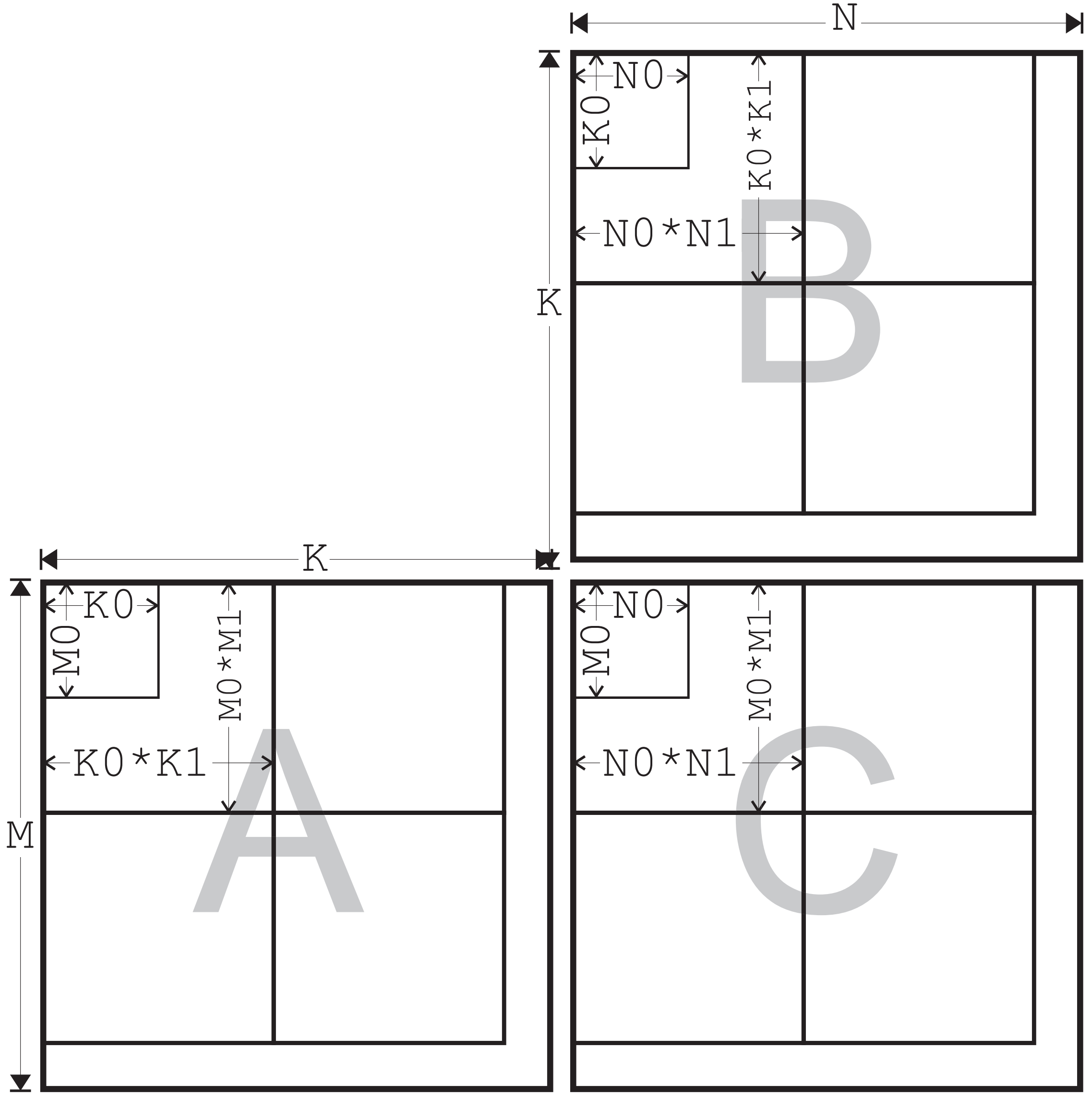
1. It is fairly easy to benchmark a large chunk of the routine's design space so near peak performance is more likely to be found. A generator can be *performance portable*.
2. The routine's development cost is amortized over a large number of platforms.
3. We have found producing a generator easier than explicit hand coding and tuning.

Search Engines

We write scripts that search a generator's entire parameter space benchmarking the resulting routines to find an optimal parameter set for a particular system. The use of search scripts can answer several questions:

- What is the fastest routine for a given workload (i.e., set of matrix sizes and their relative frequencies) under a particular compiler/machine combination?
- What is the fastest general routine across a wide range of different matrix sizes and/or different compiler/machine combinations.

Matrix Multiply Parameter Definitions



Matrix-Multiply Code Generator

The PHiPAC matrix-multiply code generator produces code for:

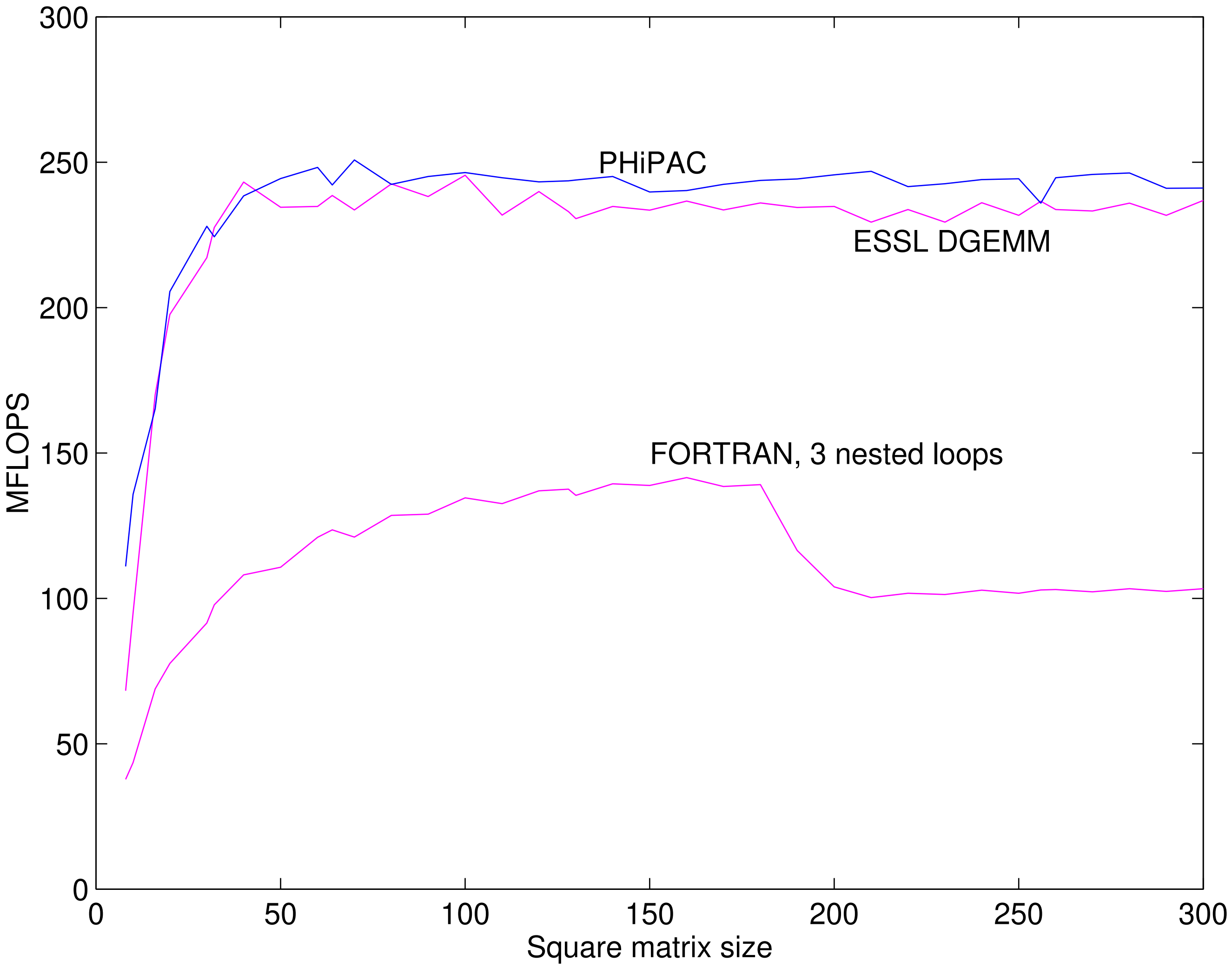
- The operation $C = \alpha op(A) op(B) + \beta C$
- Arbitrary register and cache block sizes.
- Arbitrary levels of cache blocking depth.
- Several software pipelining options: [LMA], [LM][A], [L][MA], [L][M][A]
- Different transpositions, precisions, etc.
- Called as `mm_gen -cb M0 K0 N0 [ -cb M1 K1 N1 ] ...`

Matrix-Multiply Parameter Search

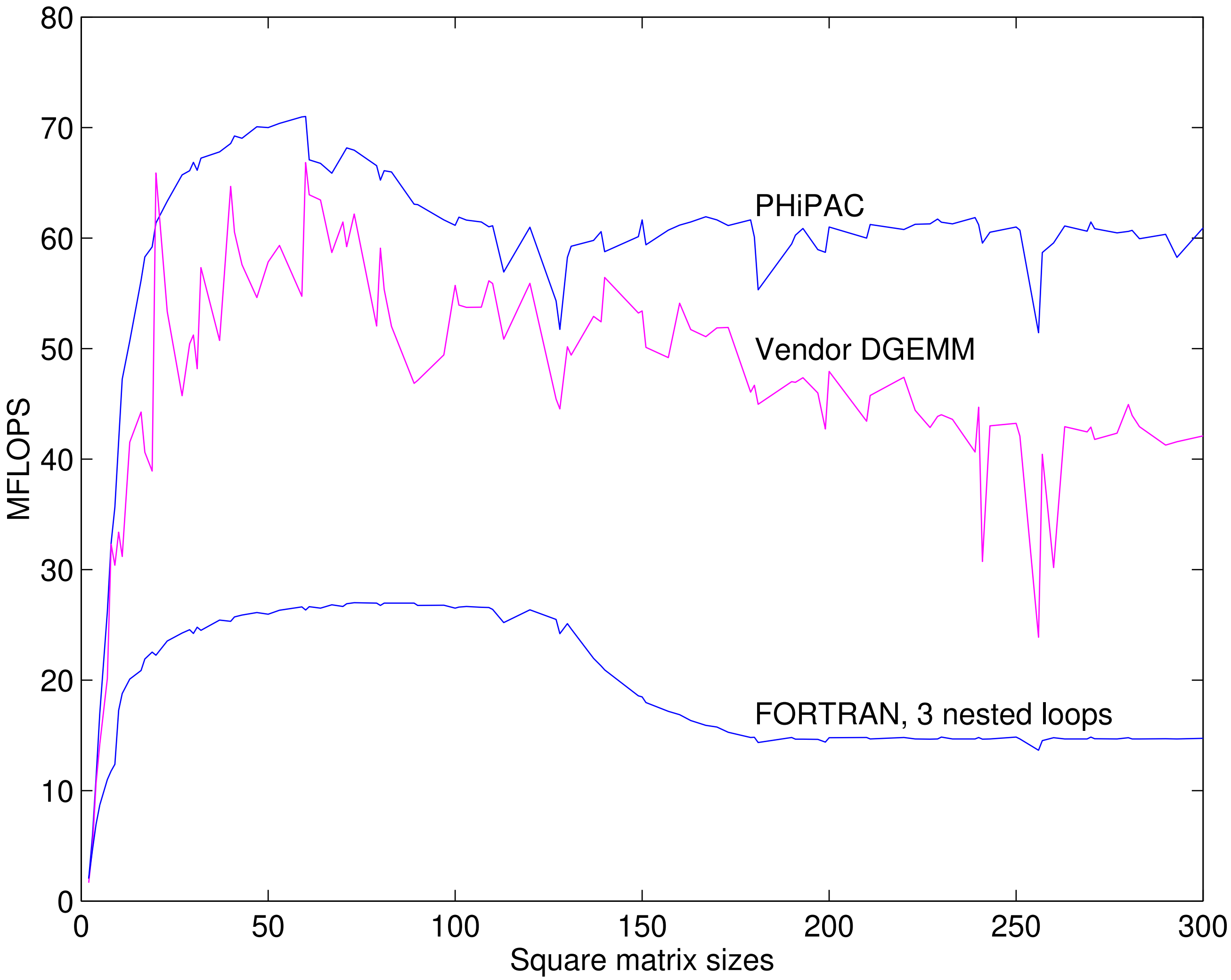
Our matrix multiply search script searches the register blocking and the L1 and L2 cache-blocking parameter space.

- Register Block Search: Search all register blocking sizes within limits.
- L1-cache Block Search: Currently, we search a small neighborhood near the $D \times D$ square case, where $3D^2 =$ L1-size.
- L2-cache Block Search: Similar to L1-cache search.

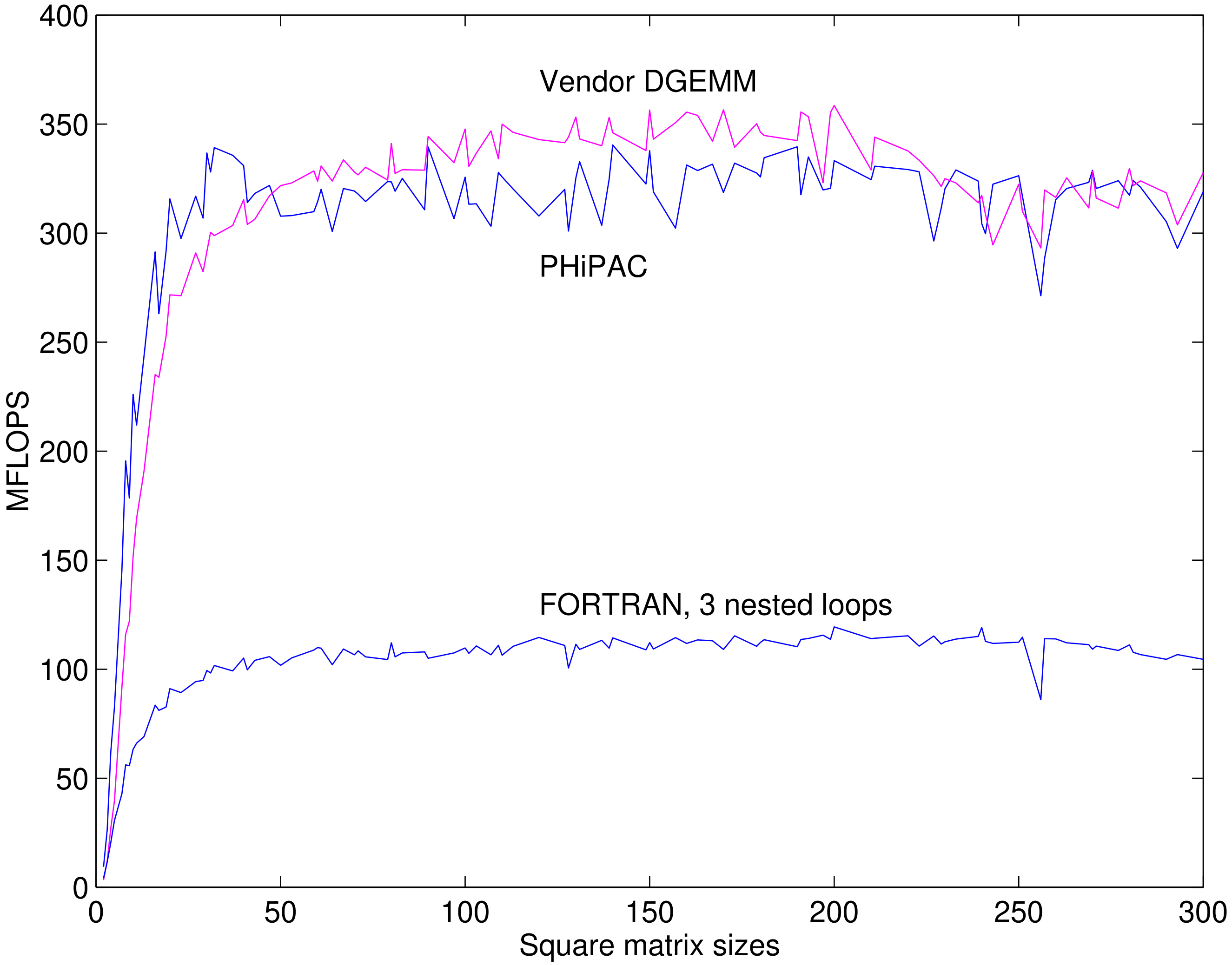
Performance on IBM RS/6000-590



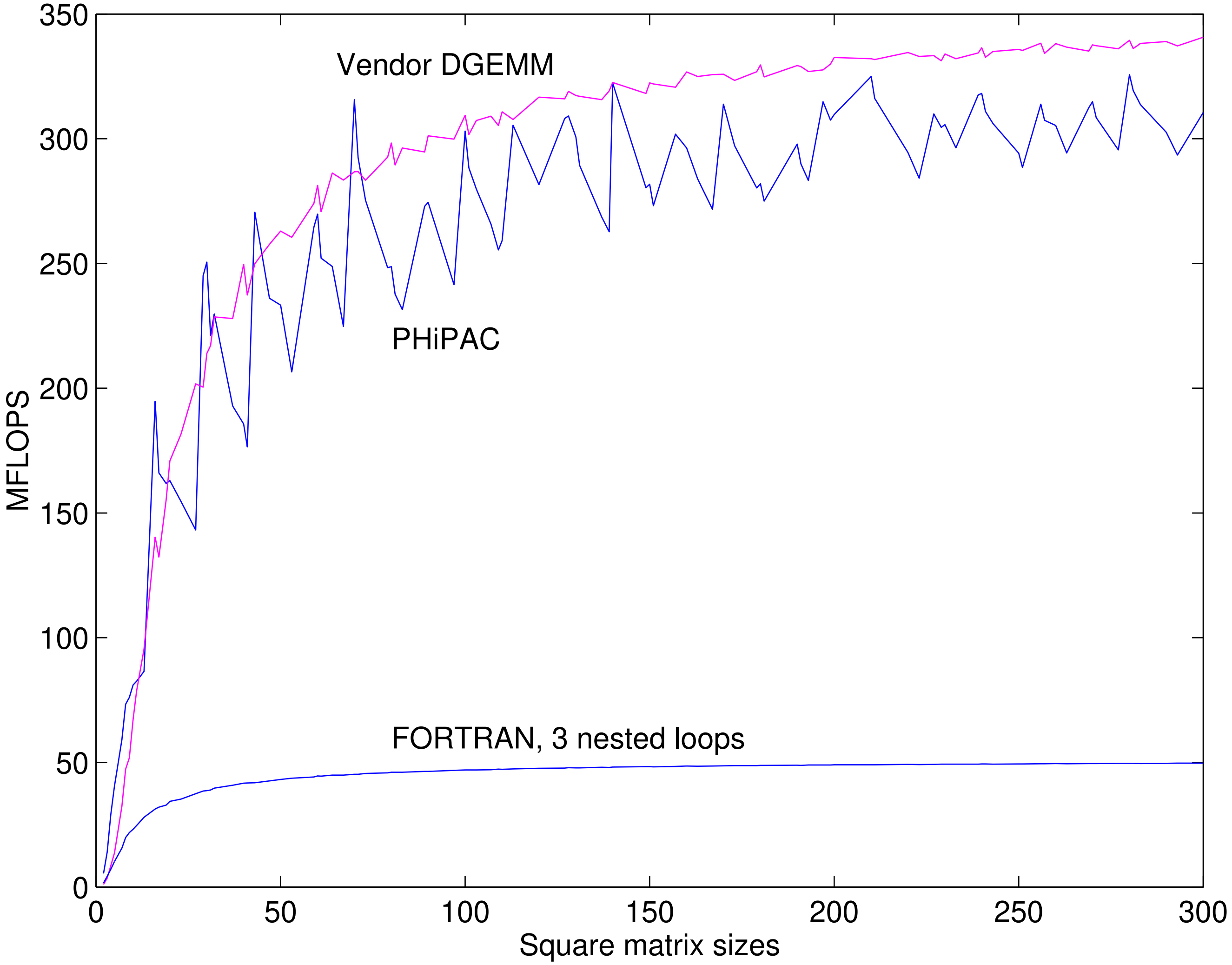
Performance on HP 712/80i



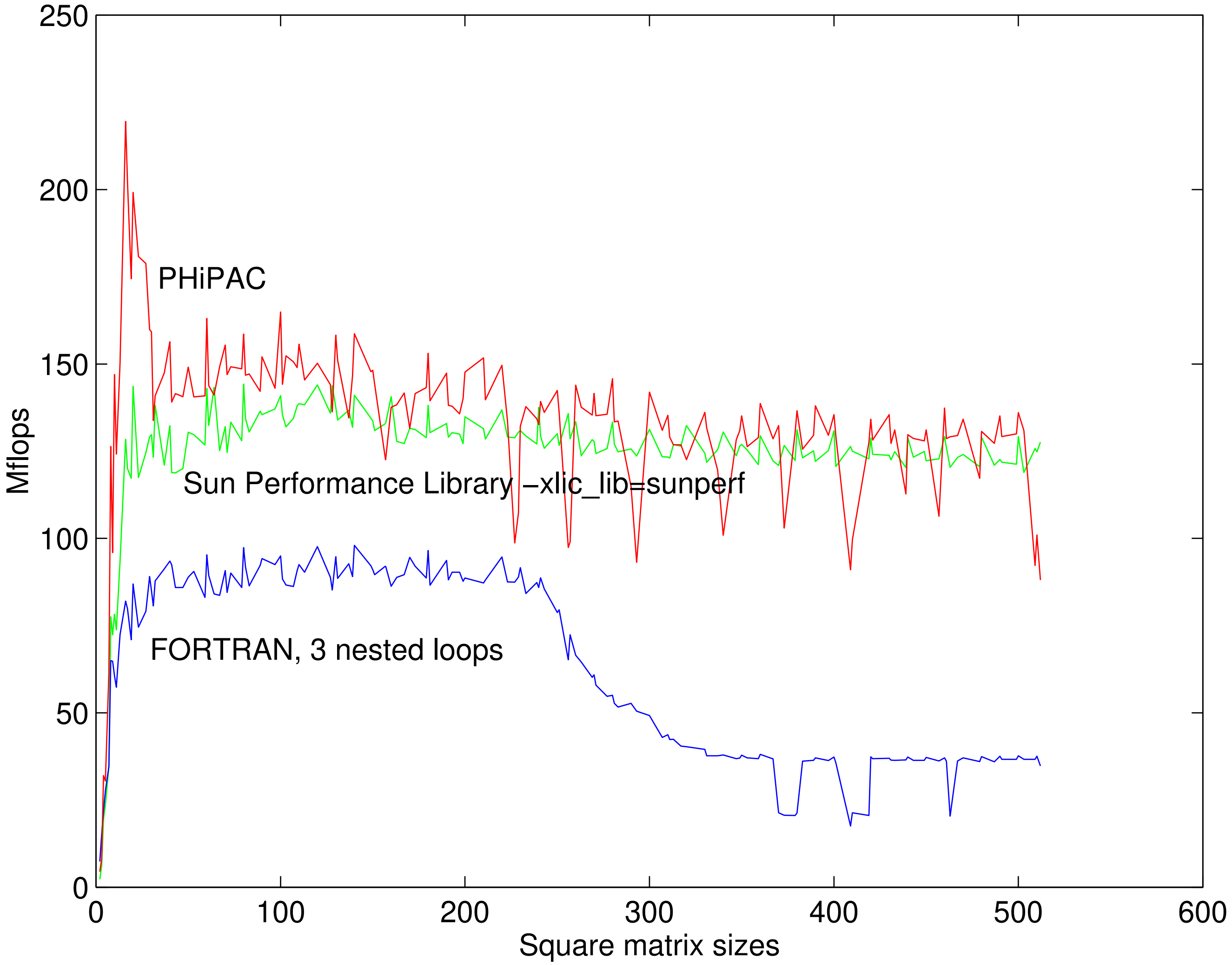
Preliminary Performance on SGI R10k Octane



Preliminary Performance SGI R8k Power Challenge



Preliminary Performance on SUN Ultra-1/170



Future Work

- A better strategy for producing performance-portable L1 and L2 cache blocked code.
- Generators for other routines such as FFT, and the other BLAS operations.

Summary and Status

- We can write portable high-performance matrix multiply code in ANSI C using careful coding, parameterized generators, and search scripts.
- The current distribution is available on the web at: `http://www.icsi.berkeley.edu/~bilmes/phipac`.
- A new distribution with both faster matrix-multiply code and search will be available at the end of this summer.