Optimizing Matrix Multiply 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

CS Division, University of California at Berkeley
Berkeley CA, 94720

International Computer Science Institute
Berkeley CA, 94704

Abstract

Modern microprocessors can achieve high performance on linear algebra kernels but this currently requires extensive machine-specific hand tuning. We have developed a methodology whereby near-peak performance on a wide range of systems can be achieved automatically for such routines. First, by analyzing current machines and C compilers, we've developed guidelines for writing Portable, High-Performance, ANSI C (PHiPAC, pronounced "fee-pack"). Second, rather than code by hand, we produce parameterized code generators. Third, we write search scripts that find the best parameters for a given system. We report on a BLAS GEMM compatible multi-level cache-blocked matrix multiply generator which produces code that achieves around 90% of peak on the Sparcstation-20/61, IBM RS/6000-590, HP 712/80i, SGI Power Challenge R8k, and SGI Octane R10k, and over 80% of peak on the SGI Indigo R4k. The resulting routines are competitive with vendor-optimized BLAS GEMMs.

[*]CS Division, University of California at Berkeley and the International Computer Science Institute. The author acknowledges the support of JSEP contract F49620-94-C-0038.

[†]CS Division, University of California at Berkeley and the International Computer Science Institute. The author acknowledges the support of ONR URI Grant N00014-92-J-1617.

[‡]CS Division, University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02).

[§]CS Division and Mathematics Dept., University of California at Berkeley. The author acknowledges the support of ARPA contract DAAL03-91-C-0047 (University of Tennessee Subcontract ORA4466.02), ARPA contract DAAH04-95-1-0077 (University of Tennessee Subcontract ORA7453.02), DOE grant DE-FG03-94ER25219, DOE contract W-31-109-Eng-38, NSF grant ASC-9313958, and DOE grant DE-FG03-94ER25206.

To appear in the proceedings of the International Conference on Supercomputing, July 1997, Vienna, Austria

1 Introduction

The use of a standard linear algebra library interface, such as BLAS [LHKK79, DCHH88, DCDH90], enables portable application code to obtain high-performance provided that an optimized library (e.g., [AGZ94, KHM94]) is available and affordable.

Developing an optimized library, however, is a difficult and time-consuming task. Even excluding algorithmic variants such as Strassen's method [BLS91] for matrix multiplication, these routines have a large design space with many parameters such as blocking sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction schedules. Furthermore, these parameters have complicated interactions with the increasingly sophisticated microarchitectures of new microprocessors.

Various strategies can be used to produced optimized routines for a given platform. For example, the routines could be manually written in assembly code, but fully exploring the design space might then be infeasible, and the resulting code might be unusable or sub-optimal on a different system.

Another commonly used method is to code using a high level language but with manual tuning to match the underlying architecture [AGZ94, KHM94]. While less tedious than coding in assembler, this approach still requires writing machine specific code which is not performance-portable across a range of systems.

Ideally, the routines would be written once in a high-level language and fed to an optimizing compiler for each machine. There is a large literature on relevant compiler techniques, many of which use matrix multiplication as a test case [WL91, LRW91, MS95, ACF95, CFH95, SMP$^+$96][1]. While these compiler heuristics generate reasonably good code in general, they tend not to generate near-peak code for any one operation. A high-level language's semantics might also obstruct aggressive compiler optimizations. Moreover, it takes significant time and investment before compiler research appears in production compilers, so these capabilities are often unavailable. While both microarchitectures and compilers will improve over time, we expect it will

[1]A longer list appears in [Wol96].

be many years before a single version of a library routine can be compiled to give near-peak performance across a wide range of machines.

We have developed a methodology, named PHiPAC, for developing Portable High-Performance linear algebra libraries in ANSI C. Our goal is to produce, with minimal effort, high-performance linear algebra libraries for a wide range of systems. The PHiPAC methodology has three components. First, we have developed a generic model of current C compilers and microprocessors that provides guidelines for producing portable high-performance ANSI C code. Second, rather than hand code particular routines, we write parameterized generators [ACF95, MS95] that produce code according to our guidelines. Third, we write scripts that automatically tune code for a particular system by varying the generators' parameters and benchmarking the resulting routines.

We have found that writing a parameterized generator and search scripts for a routine takes less effort than hand-tuning a single version for a single system. Furthermore, with the PHiPAC approach, development effort can be amortized over a large number of platforms. And by automatically searching a large design space, we can discover winning yet unanticipated parameter combinations.

Using the PHiPAC methodology, we have produced a portable, BLAS-compatible matrix multiply generator. The resulting code can achieve over 90% of peak performance on a variety of current workstations, and is sometimes faster than the vendor-optimized libraries. We focus on matrix multiplication in this paper, but we have produced other generators including dot-product, AXPY, and convolution, which have similarly demonstrated portable high performance.

We concentrate on producing high quality uniprocessor libraries for microprocessor-based systems because multiprocessor libraries, such as [CDD+96], can be readily built from uniprocessor libraries. For vector and other architectures, however, our machine model would likely need substantial modification.

Section 2 describes our generic C compiler and microprocessor model, and develops the resulting guidelines for writing portable high-performance C code. Section 3 describes our generator and the resulting code for a BLAS-compatible matrix multiply. Section 4 describes our strategy for searching the matrix multiply parameter space. Section 5 presents results on several architectures comparing against vendor-supplied BLAS GEMM. Section 6 describes the availability of the distribution, and discusses future work.

2 PHiPAC

By analyzing the microarchitectures of a range of machines, such as workstations and microprocessor-based SMP and MPP nodes, and the output of their ANSI C compilers, we derived a set of guidelines that help us attain high performance across a range of machine and compiler combinations [BAD+96].

From our analysis of various ANSI C compilers, we determined we could usually rely on reasonable register allocation, instruction selection, and instruction scheduling. More sophisticated compiler optimizations, however, including pointer alias disambiguation, register and cache blocking, loop unrolling, and software pipelining, were either not performed or not very effective at producing the highest quality code.

Although it would be possible to use another target language, we chose ANSI C because it provides a low-level, yet portable, interface to machine resources, and compilers are widely available. One problem with our use of C is that we must explicitly work around pointer aliasing as described below. In practice, this has not limited our ability to extract near-peak performance.

We emphasize that for both microarchitectures and compilers we are determining a *lowest common denominator*. Some microarchitectures or compilers will have superior characteristics in certain attributes, but, if we code assuming these exist, performance will suffer on systems where they do not. Conversely, coding for the lowest common denominator should not adversely affect performance on more capable platforms.

For example, some machines can fold a pointer update into a load instruction while others require a separate add. Coding for the lowest common denominator dictates replacing pointer updates with base plus constant offset addressing where possible. In addition, while some production compilers have sophisticated loop unrolling and software pipelining algorithms, many do not. Our search strategy (Section 4) empirically evaluates several levels of explicit loop unrolling and depths of software pipelining. While a naive compiler might benefit from code with explicit loop unrolling or software pipelining, a more sophisticated compiler might perform better without either.

2.1 PHiPAC Coding Guidelines

The following paragraphs exemplify PHiPAC C code generation guidelines. Programmers can use these coding guidelines directly to improve performance in critical routines while retaining portability, but this does come at the cost of less maintainable code. This problem is mitigated in the PHiPAC approach, however, by the use of parameterized code generators.

Use local variables to explicitly remove false dependencies.

Casually written C code often over-specifies operation order, particularly where pointer aliasing is possible. C compilers, constrained by C semantics, must obey these over-specifications thereby reducing optimization potential. We therefore remove these extraneous dependencies.

For example, the following code fragment contains a false Read-After-Write hazard:

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

The compiler may not assume `&a[i] != &b[i+1]` and is forced to first store `a[i]` to memory before loading `b[i+1]`. We may re-write this with explicit loads to local variables:

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

The compiler can now interleave execution of both original statements thereby increasing parallelism.

Exploit multiple integer and floating-point registers.

We explicitly keep values in local variables to reduce memory bandwidth demands. For example, consider the following 3-point FIR filter code:

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

The compiler will usually reload the filter values every loop iteration because of potential aliasing with `res`. We can remove the alias by preloading the filter into local variables that may be mapped into registers:

```
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++;   }
```

Minimize pointer updates by striding with constant offsets.

We replace pointer updates for strided memory addressing with constant array offsets. For example:

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

should be converted to:

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

Compilers can fold the constant index into a register plus offset addressing mode.

Hide multiple instruction FPU latency with independent operations.

We use local variables to expose independent operations so they can be executed independently in a pipelined or superscalar processor. For example:

```
f1=f5*f9;f2=f6+f10;f3=f7*f11;f4=f8+f12;
```

Balance the instruction mix.

A balanced instruction mix has a floating-point multiply, a floating-point add, and 1–2 floating-point loads or stores interleaved. It is not worth decreasing the number of multiplies at the expense of additions if the total floating-point operation count increases.

Increase locality to improve cache performance.

Cached machines benefit from increases in spatial and temporal locality. Whenever possible, we arrange our code to have predominantly unit-stride memory accesses, and try to reuse data once it is in cache. See Section 3.1, for our blocked matrix multiply example.

Convert integer multiplies to adds.

Integer multiplies and divides are slow relative to integer addition. Therefore, we use pointer updates rather than subscript expressions. Rather than:

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

we produce:

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

Minimize branches, avoid magnitude compares.

Branches are costly, especially on modern superscalar processors. Therefore, we unroll loops to amortize branch cost and use C `do {} while ();` loop control whenever possible to avoid any unnecessary compiler-produced loop head branches.

Also, on many microarchitectures, it is cheaper to perform equality or inequality loop termination tests than magnitude comparisons. For example, 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);
```

This also removes one loop control variable.

Loop unroll explicitly to expose optimization opportunities.

We unroll loops explicitly to increase opportunities for other performance optimizations. For example, our 3-point FIR filter example above may be further optimized as follows:

```
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 ( ... );
```

In the inner loop, there are now only two memory access per five floating point operations whereas in our unoptimized code, there were seven memory accesses per five floating point operations.

3 Matrix Multiply Generator

mm_gen is a generator that produces C code, following the PHiPAC coding guidelines, for one variant of the matrix multiply operation $C = \alpha op(A)op(B) + \beta C$ where $op(A)$, $op(B)$, and C, are respectively M×K, K×N, and M×N matrices, α and β are scalar parameters, and $op(X)$ is either $transpose(X)$ or just X. Our individual procedures have a lower level interface then a BLAS GEMM and have no error checking. For optimal efficiency, error checking should be performed by

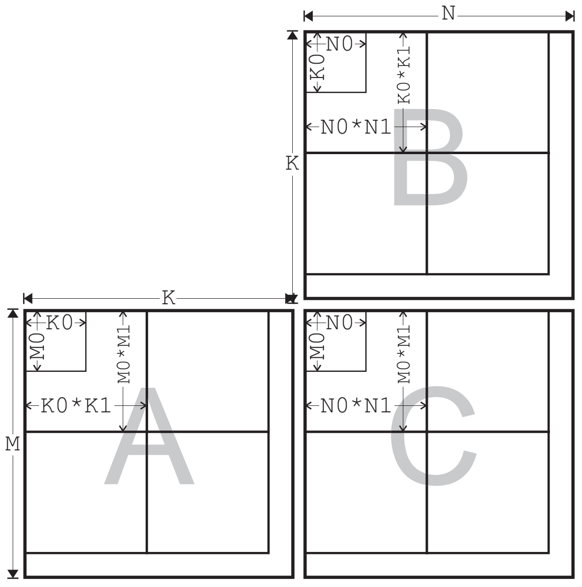

Figure 1: Matrix blocking parameters

the caller when necessary rather than unnecessarily by the callee. We create a full BLAS-compatible GEMM, by generating all required matrix multiply variants and linking with our GEMM-compatible interface that includes error checking.

mm_gen produces a cache-blocked matrix multiply [GL89, LRW91, MS95], restructuring the algorithm for unit stride, and reducing the number of cache misses and unnecessary loads and stores. Under control of command line parameters, **mm_gen** can produce blocking code for any number of levels of memory hierarchy, including register, L1 cache, TLB, L2 cache, and so on. **mm_gen**'s code can also perform copy optimization [LRW91], optionally with a different accumulator precision. The latest version can also generate the innermost loop with various forms of software pipelining.

A typical invocation of **mm_gen** is:

```
mm_gen -cb M0 K0 N0 [ -cb M1 K1 N1 ] ...
```

where the register blocking is M_0, K_0, N_0, the L1-cache blocking is M_1, K_1, N_1, etc. The parameters M_0, K_0, and N_0 are specified in units of matrix elements, i.e., single, double, or extended precision floating-point numbers, M_1, K_1, N_1 are specified in units of register blocks, M_2, K_2, and K_2 are in units of L1 cache blocks, and so on. For a particular cache level, say i, the code accumulates into a C destination block of size $M_i \times N_i$ units and uses A source blocks of size $M_i \times K_i$ units and B source blocks of size $K_i \times N_i$ units (see Figure 1).

3.1 Matrix Multiply Code

In this section, we examine the code produced by **mm_gen** for the operation `C = C + A*B` where A (respectively B, C) is an M×K (respectively K×N, M×N) matrix. Figure 2 lists the L1 cache blocking core code comprising the 3 nested loops, M, N, and K. **mm_gen** does not vary the loop permutation [MS95, LRW91] because the resulting gains in locality are subsumed by the method described below.

The outer M loop in Figure 2 maintains pointers `c0` and `a0` to rows of register blocks in the A and C matrices. It also maintains end pointers (`ap0_endp` and `cp0_endp`) used for loop termination. The middle N loop maintains a pointer `b0` to columns of register blocks in the B matrix, and maintains a pointer `cp0` to the current C destination register block. The N loop also maintains separate pointers (`ap0_0` through `ap0_1`) to successive rows of the current A source block. It also initializes a pointer `bp0` to the current B source block. We assume local variables can be held in registers, so our code uses many pointers to minimize both memory references and integer multiplies.

The K loop iterates over source matrix blocks and accumulates into the same $M_0 \times N_0$ destination block. We assume that the floating-point registers can hold a $M_0 \times N_0$ accumulator block, so this block is loaded once before the K loop begins and stored after it ends. The K loop updates the set of pointers to the A source block, one of which is used for loop termination.

The parameter K_0 controls the extent of inner loop unrolling as can be seen in Figure 2. The unrolled core loop performs K_0 outer products accumulating into the C destination block. We code the outer products by loading one row of the B block, one element of the A block, then performing N_0 multiply-accumulates. The C code uses $N_0 + M_0$ memory references per $2N_0M_0$ floating-point operations in the inner K loop, while holding $M_0N_0 + N_0 + 1$ values in local variables. While the intent is that these local variables map to registers, the compiler is free to reorder all of the independent loads and multiply-accumulates to trade increased memory references for reduced register usage. The compiler also requires additional registers to name intermediate results propagating through machine pipelines.

The code we have so far described is valid only when M, K, and N are integer multiples of M_0, K_0, and N_0 respectively. In the general case, **mm_gen** also includes code that operates on power-of-two sized fringe strips, i.e., 2^0 through $2^{\lfloor \log_2 L \rfloor}$ where L is M_0, K_0, or N_0. We can therefore manage any fringe size from 1 to L−1 by executing an appropriate combination of fringe code. The resulting code size growth is logarithmic in the register blocking (i.e., $O(\log(M_0)\log(K_0)\log(N_0))$) yet maintains good performance. To reduce the demands on the instruction cache, we arrange the code into several independent sections, the first handling the matrix core and the remainder handling the fringes.

The latest version of the generator can optionally produce code with a software-pipelined inner loop. Each outer product consists of a load, a multiply, and an accumulate section. We group these sections into three software pipelined code variants: two two-stage pipes with stages [load-multiply, accumulate] and [load, multiply-accumulate], and a three-stage pipe with stages [load, multiply, accumulate]. Recent results show that this software pipelining can result in an appreciable speedup.

Because of the separation between matrix dimension and matrix stride, we can implement higher levels of cache blocking as calls to lower level routines with appropriately sized sub-matrices.

```
mul_mfmf_mf(const int M,const int K,const int N,
const float*const A,const float*const B,float*const C,
const int Astride,const int Bstride,const int Cstride)
{
   const float *a,*b; float *c;
   const float *ap_0,*ap_1; const float *bp; float *cp;
   const int A_sbs_stride = Astride*2;
   const int C_sbs_stride = Cstride*2;
   const int k_marg_el = K & 1;
   const int k_norm = K - k_marg_el;
   const int m_marg_el = M & 1;
   const int m_norm = M - m_marg_el;
   const int n_marg_el = N & 1;
   const int n_norm = N - n_marg_el;
   float *const c_endp = C+m_norm*Cstride;
   register float c0_0,c0_1,c1_0,c1_1;
   c=C;a=A;
   do { /* M loop */
      const float* const ap_endp = a + k_norm;
      float* const cp_endp = c + n_norm;
      const float* const apc_1 = a + Astride;
      b=B;cp=c;
      do {  /* N loop */
         register float _b0,_b1;
         register float _a0,_a1;
         float *_cp;
         ap_0=a;ap_1=apc_1;bp=b;
         _cp=cp;c0_0=_cp[0];c0_1=_cp[1];
         _cp+=Cstride;c1_0=_cp[0];c1_1=_cp[1];
         do { /* K loop */
            _b0 = bp[0]; _b1 = bp[1];
            bp += Bstride; _a0 = ap_0[0];
            c0_0 += _a0*_b0; c0_1 += _a0*_b1;
            _a1 = ap_1[0];
            c1_0 += _a1*_b0; c1_1 += _a1*_b1;

            _b0 = bp[0]; _b1 = bp[1];
            bp += Bstride; _a0 = ap_0[1];
            c0_0 += _a0*_b0; c0_1 += _a0*_b1;
            _a1 = ap_1[1];
            c1_0 += _a1*_b0; c1_1 += _a1*_b1;

            ap_0+=2;ap_1+=2;
         } while (ap_0!=ap_endp);
         _cp=cp;_cp[0]=c0_0;_cp[1]=c0_1;
         _cp+=Cstride;_cp[0]=c1_0;_cp[1]=c1_1;
         b+=2;cp+=2;
      } while (cp!=cp_endp);
      c+=C_sbs_stride;a+=A_sbs_stride;
   } while (c!=c_endp);
}
```

Figure 2: $M_0 = 2$, $K_0 = 2$, $N_0 = 2$ matrix multiply L1 routine for $M \in \{2m : m \geq 1\}, K \in \{2k : k \geq 1\}, N \in \{2n : n \geq 1\}$. Within the K-loop is our fully-unrolled $2 \times 2 \times 2$ core matrix multiply. The code is not unlike the register code in [CFH95]. In our terminology, the leading dimensions LDA, LDB, and LDC are called `Astride`, `Bstride`, and `Cstride` respectively. The four local variables `c0_0` through `c1_1` hold a complete C destination block. Variables `ap_0` and `ap_1` point to successive rows of the A source matrix block, and variable `bp` points to the first row of the B source matrix block. Elements in A and B are accessed using constant offsets from the appropriate pointers.

4 Matrix Multiply Search Scripts

The search script take parameters describing the machine architecture, including the number of integer and floating-point registers and the sizes of each level of cache. For each combination of generator parameters and compilation options, the matrix multiply search script calls the generator, compiles the resulting routine, links it with timing code, and benchmarks the resulting executable.

To produce a complete BLAS GEMM routine, we find separate parameters for each of the three cases $A \times B$, $A^T \times B$, and $A \times B^T$ ($A^T \times B^T$ has code identical to $A \times B$). For each case, we first find the best register (or L0) parameters for in-L1-cache matrices, then find the best L1 parameters for in-L2-cache matrices, etc. While this strategy is not guaranteed to find the best L0 core for out-of-L1-cache matrices, the resulting cores have performed well in practice.

4.1 Register (L0) Parameter Search

The register block search evaluates all combinations of M_0 and N_0 where $1 \leq M_0 N_0 \leq \mathrm{NR}$ and where NR is the number of machine floating-point registers. We search the above for $1 \leq K_0 \leq K_0^{max}$ where $K_0^{max} = 20$ but is adjustable. Empirically, $K_0^{max} > 20$ has never shown appreciable benefit.

Our initial strategy [BAD+96] benchmarked a set of square matrices that fit in L1 cache. We then chose the L0 parameters that achieved the highest performance. While this approach gave good performance, the searches were time consuming.

We noted that that the majority of the computation, especially for larger matrices, is performed by the core $M_0 \times K_0 \times N_0$ register blocked code. Our newer search strategy, therefore, produces code containing only the core, which decreases compile time, and for each L0 parameter set, we benchmark only a single matrix multiply with size $M = M_0$, $N = N_0$, and $K = kK_0$. The parameter k is chosen such that the three matrices are no larger than L1 cache (we call this a "fat" dot-product). While this case ignores the cost of the M- and N-loop overhead, we have so far found this approach to produce comparable quality code in much less time than the previous strategy (e.g., 5 hours vs. 24 hours on the SGI Indigo R4K).

We initially thought the order in which we searched the L0 parameter space could have an effect on search time and evaluated orders such as i-j-k, random, best-first, and simulated annealing. We found, however, that the added search complexity outweighed any benefit so we settled on a random strategy.

4.2 Cache Block Search

We perform the L1 cache blocking search after the best register blocking is known. We would like to make the L1 blocks large to increase data reuse but larger L1 blocks increase the probability of cache conflicts [LRW91]. Tradeoffs between M- and N- loop overheads, memory access patterns, and TLB structure also affect the best L1 size. We currently perform a relatively simple search of the L1 parameter space. For the D×D square case, we search the neighborhood centered at $3D^2 = L1$ where $L1$ is the L1 cache size in elements. We set M_1 to the values $\lfloor \phi D / M_0 \rfloor$ where $\phi \in \{0.25, 0.5, 1.0, 1.5, 2.0\}$ and $D = \sqrt{L1/3}$. K_1 and N_1 are set similarly. We benchmark the resulting 125 combinations with matrix sizes that either fit in L2 cache, or are within some upper bound if no L2 cache exists. The L2 cache blocking search, when necessary, is performed in a similar way.

5 Results

We ran the search scripts to find the best register and L1 cache blocking parameters for six commercial workstation systems. These systems have different instruction set architectures and widely varying microarchitectures and memory hierarchies. The results are summarized in Table 1.

The SGI R8K and R10K searches used the newer version of the code generator and search scripts, while other results were obtained with the alpha release. Figures 3–6 plot the performance of the resulting routines for all square matrices $M = K = N = D$, where D runs over powers of 2 and 3, multiples of 10, and primes, up to a maximum of 300. We compare with the performance of a vendor-optimized BLAS GEMM where available. In each case, PHiPAC yields a substantial fraction of peak performance and is competitive with vendor BLAS. Due to limited availability, we could only perform an incomplete search on the R8K and R10K, and so these are preliminary performance numbers. There is also potential for improvement on the other machines when we rerun with the newer version. For completeness, we also show the poor performance obtained when compiling a simple three nested loop version of GEMM with FORTRAN or C optimizing compilers.

The PHiPAC methodology can also improve performance even if there is no scope for memory blocking. In Figure 9 we plot the performance of a dot product code generated using PHiPAC techniques. Although the parameters used were obtained using a short manual search, we can see that we are competitive with the assembly-coded SGI BLAS SDOT.

In some of the plots, we see that PHiPAC routines suffer from cache conflicts. Our measurements exaggerate this effect by including all power-of-2 sized matrices, and by allocating all regions contiguously in memory. For matrix multiply, we can reduce cache conflicts by copying to contiguous memory when pathological strides are encountered [LRW91]. Unfortunately, this approach does not help dot product. One drawback of the PHiPAC approach is that we can not control the order compilers schedule independent loads. We've occasionally found that exchanging two loads in the assembly output for dot product can halve the number of cache misses where conflicts occur, without otherwise impacting performance.

6 Status, Availability, and Future Work

This paper has demonstrated our ability to write portable, high performance ANSI C code for matrix multiply using parameterized code generators and a timing-driven search strategy.

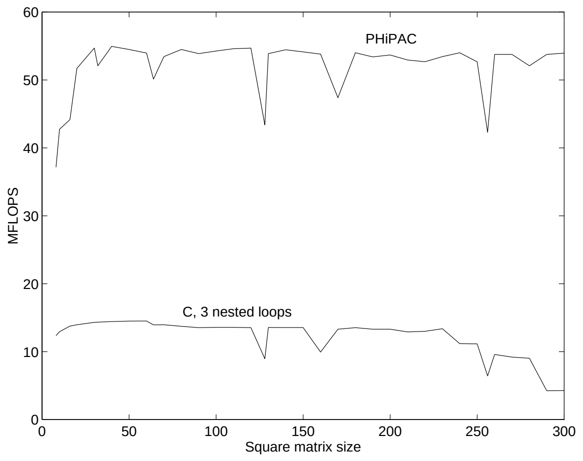

Figure 3: Performance of single precision matrix multiply on a Sparcstation-20/61.

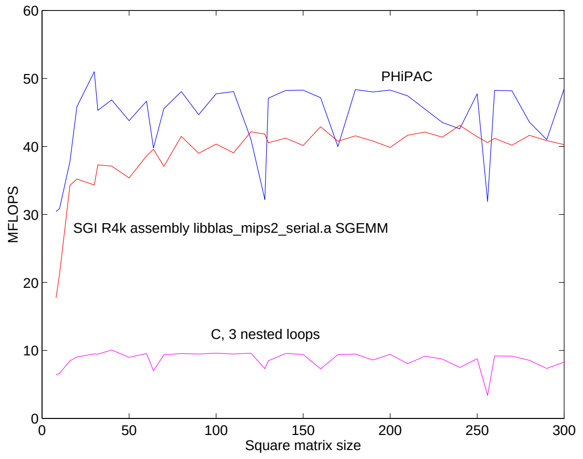

Figure 4: Performance of single precision matrix multiply on a 100 MHz SGI Indigo R4K. We show the SGEMM from SGI's `libblas_mips2_serial.a` library.

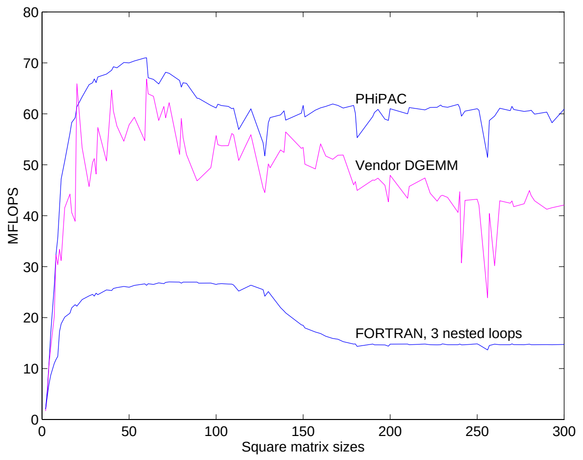

Figure 5: Performance of double precision matrix multiply on a HP 712/80i. We show DGEMM from the `pa1.1` version of `libvec.a` in HP's compiler distribution.

	Sun Sparc-20/61	HP 712/80i	IBM RS/6000-590	SGI Indigo R4K	SGI Challenge	SGI Octane
Processor	SuperSPARC+	PA7100LC	RIOS-2	R4K	R8K	R10K
Frequency (MHz)	60	80	66.5	100	90	195
Max Instructions/cycle	3	2	6	1	4	4
Peak MFLOPS (32b/64b)	60/60	160/80	266/266	67/50	360/360	390/390
FP registers (32b/64b)	32/16	64/32	32/32	16/16	32/32	32/32
L1 Data cache (KB)	16	128	256	8	-	32
L2 Data cache (KB)	1024	-	-	1024	4096	1024
OS	SunOS 4.1.3	HP-UX 9.05	AIX 3.2	Irix 4.0.5H	IRIX6.2	IRIX6.4
C Compiler	Sun acc 2.0.1	HP c89 9.61	IBM xlc 1.3	SGI cc 3.10	SGI cc	SGI cc
Search results						
PHiPAC version	alpha	alpha	alpha	alpha	new	new
Precision	32b	64b	64b	32b	64b	64b
M_0,K_0,N_0	2,4,10	3,1,2	2,1,10	2,10,3	2,4,14	4,2,6
M_1,K_1,N_1	26,10,4	30,60,30	105,70,28	30,4,10	200,80,25	12,24,9
CFLAGS	-fast	-O	-O3 -qarch=pwr2	-O2 -mips2	-n32 -mips4 -O3	

Table 1: Workstation system details and results of matrix multiply parameter search.

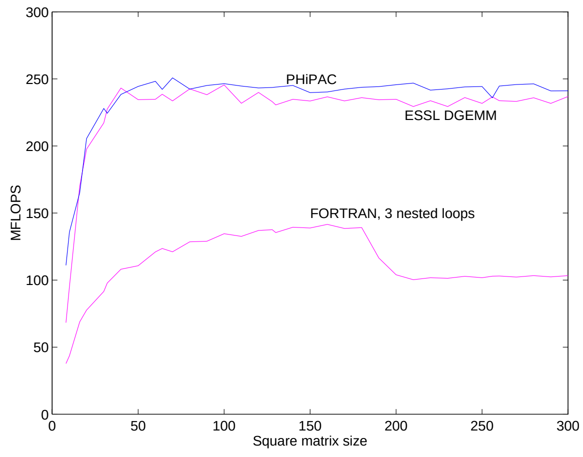

Figure 6: Performance of double precision matrix multiply on an IBM RS/6000-590. We show the DGEMM from IBM's POWER2-optimized ESSL library.

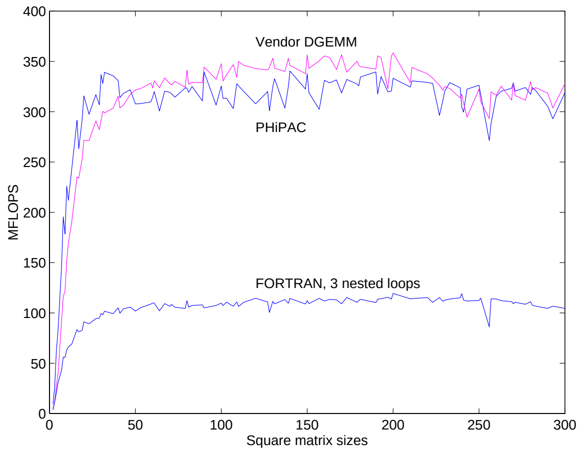

Figure 8: Preliminary performance of double precision matrix multiply on an SGI R10K Octane. We show the DGEMM from SGI's R10K-optimized `libblas`.

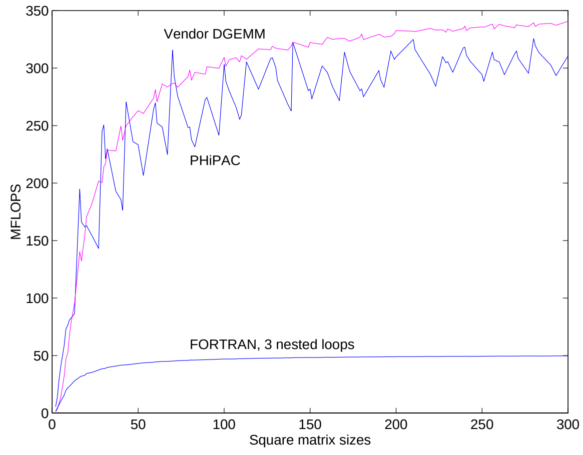

Figure 7: Preliminary performance of double precision matrix multiply on an SGI R8K Power Challenge. We show the DGEMM from SGI's R8K-optimized `libblas`.

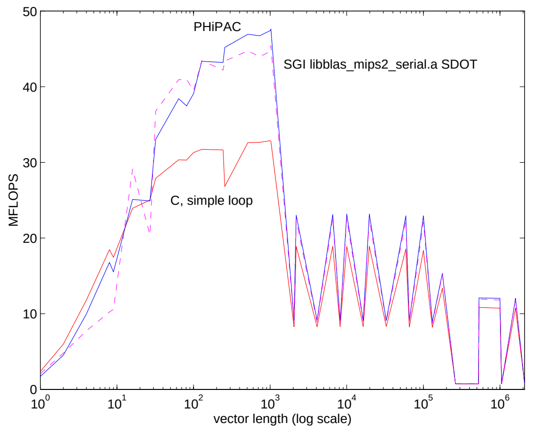

Figure 9: Performance of single precision unit-stride dot-product on a 100 MHz SGI R4K.

The PHiPAC alpha release contains the matrix multiply generator, the naive search scripts written in `perl`, and our timing libraries. We have created a Web site from which the alpha release is available and on which we plan to list blocking parameters for many systems [BAD+]. We are currently working on a better L1 blocking strategy and accompanying methods for search based on various criteria [LRW91]. The PHiPAC GEMM can be used with Bo Kågström's GEMM-based BLAS3 package [BLL93] and LAPACK [ABB+92].

We have also written parameterized generators for matrix-vector and vector-matrix multiply, dot product, AXPY, convolution, and outer-product, and further generators, such as for FFT, are planned.

We wish to thank Ed Rothberg of SGI for help obtaining the R8K and R10K performance plots. We also wish to thank Nelson Morgan who provided initial impetus for this project and Dominic Lam for work on the initial search scripts.

References

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK users' guide, release 1.0. In *SIAM*, Philadelphia, 1992.

[ACF95] B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: A methodology for portable high-performance. In *International Working Conference on Massively Parallel Programming Models*, 1995.

[AGZ94] R. Agarwal, F. Gustavson, and M. Zubair. *IBM Engineering and Scientific Subroutine Library, Guide and Reference*, 1994. Available through IBM branch offices.

[BAD+] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. The PHiPAC WWW home page. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[BAD+96] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. PHiPAC: A portable, high-performance, ANSI C coding methodology and its application to matrix multiply. LAPACK working note 111, University of Tennessee, 1996.

[BLL93] B.Kågström, P. Ling, and C. Van Loan. Portable high performance GEMM-based level 3 BLAS. In R.F. Sincovec et al., editor, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM Publications.

[BLS91] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[CDD+96] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPAC: A portable linear algebra library for distributed memory computers - design issues and performance. LAPACK working note 95, University of Tennessee, 1996.

[CFH95] L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.

[DCDH90] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[DCHH88] J. Dongarra, J. Du Cros, S. Hammarling, and R.J. Hanson. An extended set of FORTRAN basic linear algebra subroutines. *ACM Trans. Math. Soft.*, 14:1–17, March 1988.

[GL89] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[KHM94] C. Kamath, R. Ho, and D.P. Manley. DXML: A high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, Summer 1994.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[LRW91] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of ASPLOS IV*, pages 63–74, April 1991.

[MS95] J.D. McCalpin and M. Smotherman. Automatic benchmark generation for cache optimization of matrix algorithms. In R. Geist and S. Junkins, editors, *Proceedings of the 33rd Annual Southeast Conference*, pages 195–204. ACM, March 1995.

[SMP+96] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The combined effectiveness of unimodular transformations, tiling, and software prefetching. In *Proceedings of the 10th International Parallel Processing Symposium*, April 15–19 1996.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[Wol96] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley, 1996.