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

Jeff Bilmes, Krste Asanović, Chee-Whye Chin and Jim Demmel
CS Division, University of California at Berkeley, Berkeley CA
International Computer Science Institute, Berkeley CA
Proceedings of the 11th International Conference on Supercomputing (1997)

Stefan Dietiker, October 5th 2011
Matrix Multiplications
They are important & interesting

- Linear Algebra
  - LA-Kernels, such as LAPACK, heavily use Matrix Multiplication
  - There are numerous vendor optimized BLAS-libraries

- Computational viewpoint
  - A lot of potential for code optimization
Traditional Approach
Hand-optimized libraries

Diagram:
- Programmer
  - Source Code
  - Compiler
    - Optimized Library (x86)
      - Executable
    - Optimized Library (ARM)
      - Executable

x86
ARM
Traditional Approach
Hand-optimized libraries
Traditional Approach
Hand-optimized libraries

• **In general**: (Micro-)Architecture specific code is unportable.

• Assembler code is difficult to write and maintain. => High Effort

• We prefer to write code in a **high level standardized** language that can be compiled on many different platforms.
PHiPAC Approach
Generate optimized source code
PHiPAC Approach
Parameters are architecture specific
PHiPAC Approach
Look ahead

Source: PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology
Coding Guidelines
Remove false dependencies

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

? \&a[i] == \&b[i+1]
Coding Guidelines
Remove false dependencies

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

```c
&a[i] != &b[i+1]
```

![Program flow diagram](image)
Coding Guidelines
Scalar Replacement: Exploit Register File

```c
while(...) {
    *res++ = f[0] * sig[0] +
             f[1] * sig[1] +
             f[2] * sig[2];
    sig++; }
```

```c
float f0,f1,f2;
f0=f[0];f1=f[1];f2=[2];
while(...) {
    *res++ = f0*sig[0] +
             f1*sig[1] +
             f2*sig[2];
    sig++;}
```
Coding Guidelines
Minimize pointer updates

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

```asm
movl (%ecx), %eax
addl $16, %ecx
movl (%ecx), %ebx
addl $16, %ecx
movl (%ecx), %edx
addl $16, %ecx
movl (%ecx), %esi
addl $16, %ecx
```

(IA32 Assembler)

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

```asm
movl (%ecx), %eax
movl 16(%ecx), %ebx
movl 32(%ecx), %edx
movl 48(%ecx), %esi
```

(IA32 Assembler)
Coding Guidelines
Improve temporal and spatial locality

- **Temporal locality**: The delay between two consecutive memory accesses to the same memory location should be as short as possible.

- **Spatial locality**: Consecutive operations should access the same memory area.
## Coding Guidelines

### Summary

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Effect</th>
<th>Parameterizable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Scalar Replacement to remove false dependencies</td>
<td>Parallel execute of independent operations</td>
<td></td>
</tr>
<tr>
<td>Use Scalar Replacement exploit register file</td>
<td>Decreased memory bandwidth</td>
<td>yes</td>
</tr>
<tr>
<td>Use Scalar Replacement minimize pointer updates</td>
<td>Compressed instruction sequence</td>
<td></td>
</tr>
<tr>
<td>Hide multiple instruction FPU latency</td>
<td>Independent execution of instructions in pipelined CPUs</td>
<td></td>
</tr>
<tr>
<td>Balance the instruction mix</td>
<td>Increased instruction throughput</td>
<td></td>
</tr>
<tr>
<td><strong>Increase locality</strong></td>
<td>Increased cache performance</td>
<td>yes</td>
</tr>
<tr>
<td>Minimize branches</td>
<td>Decrease number of pipeline flushes</td>
<td></td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Compressed instruction sequence</td>
<td>yes</td>
</tr>
<tr>
<td>Convert integer multiplies to adds</td>
<td>Decrease instruction latency</td>
<td></td>
</tr>
</tbody>
</table>
Matrix Multiplications
Simplest Approach: Three nested loops

```c
for (i=0; i<M; i++)
    for (j=0; j<N; j++)
        for (l=0; l<K; l++)
            c[i][j] += a[i][l] * b[l][j];
```
Block Matrix Multiplication
General Approach

for (i=0; i<M; i+=MBlock)
  for (j=0; j<N; j+=NBlock)
    for (l=0; l<K; l+=KBlock)
      for (r=i; r<i+MBlock; r++)
        for (s=i; s<i+NBlock; s++)
          for (t=i; t<i+KBlock; t++)
            c[r][s] += a[r][t] * b[t][s];
Matrix Multiplications
Choose appropriate block sizes

```c
for (i=0; i<M; i+=M0)
    for (j=0; j<N; j+=N0)
        for (l=0; l<K; l+=K0)
            for (r=i; r<i+M0; r++)
                for (s=i; s<i+N0; s++)
                    for (t=i; t<i+K0; t++)
                        c[r][s] += a[r][t] * b[t][s];
```
Parameterized Generator
Choose appropriate block sizes

$ mm_gen -l0 <M0> <K0> <N0> [-l1 <M1> <K1> <N1>]
Matrix Multiplications
Blocking Example: innermost 2x2 Blocks

$ mm_cgen -l0 2 2 2 -l1 4 4 4$

do { /* */
do { /*...*/
do { /*...*/
    _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(); /*...*/
Finding Optimal Block Sizes
Using a Search Script

Optimized C → Compiler → Executable

mm_gen → Search Script → Benchmark

Architecture Specific Parameters
Finding Optimal Block Sizes

Example: Finding the L1 Parameters

- We have to limit the parameter space
- For the square case $D \times D$
- We search the neighborhood centered at $3D^2 = L_1$
- We set $M_1, K_1, N_1$ to the values $\phi \frac{D}{M_0}$
- Where $\phi \in (0.25, 0.5, 1.0, 1.5, 2.0)$
- $\Rightarrow$ 125 Combinations
Results

Example (Single Precision Matrix Mult. on a 100MHz SGI Indigo R4K)

Source: PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology
Results
Example (Double Precision Matrix Mult. on a SGI R8K Power Challenge)

Source: PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology
Strengths & Limitations
There's no golden hammer

• Strengths
  • **Automatic** Search for optimal Parameters
  • Produces **portable** ANSI C Code.

• Limitations
  • Focus on **uniprocessor** Machines
  • No support for **vector** based CPUs
  • No control over instruction scheduling
Further Information
Try yourself…

- Website:
  http://www.icsi.berkeley.edu/~bilmes/phipec/