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

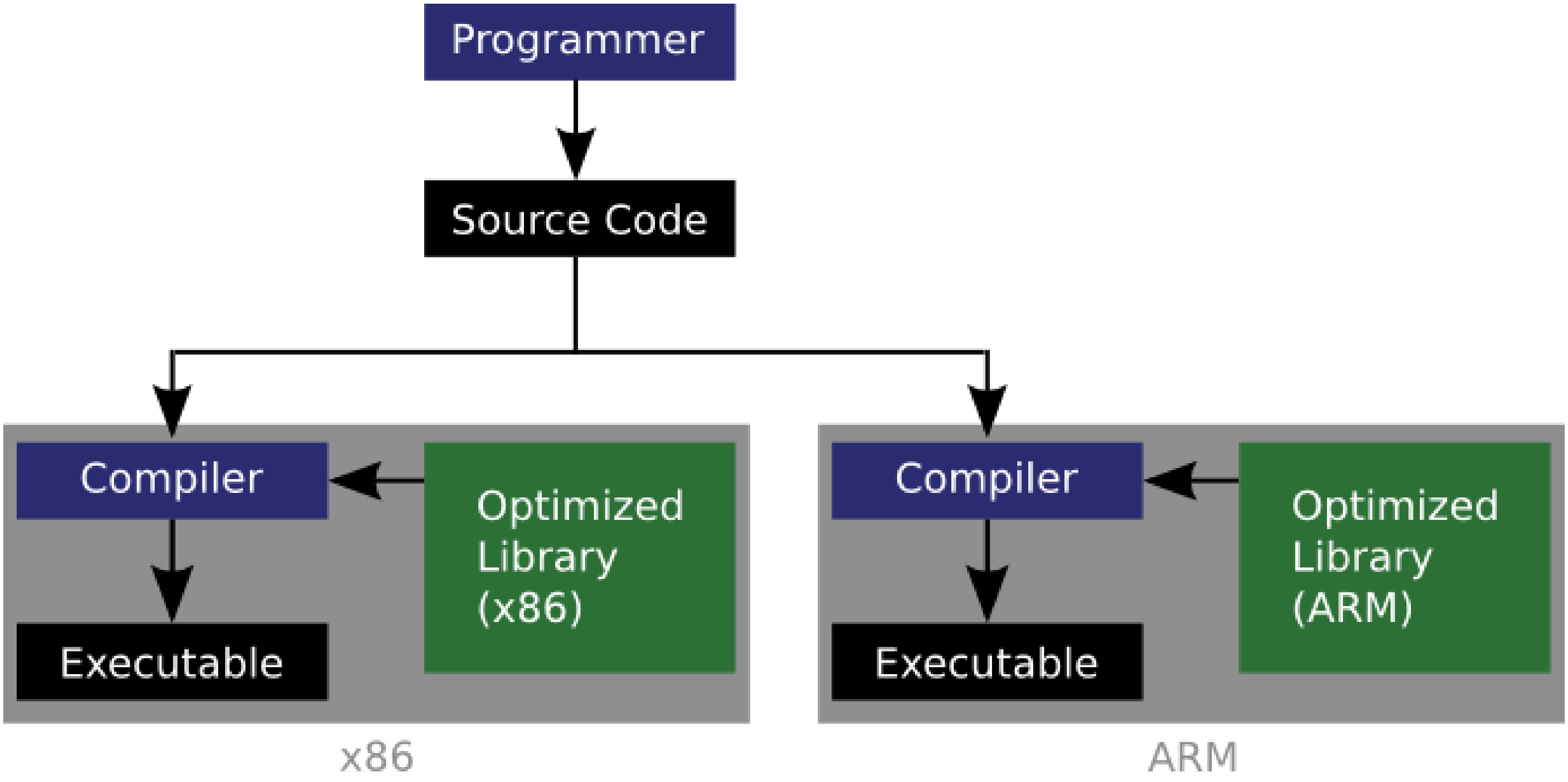
Matrix Multiplications

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

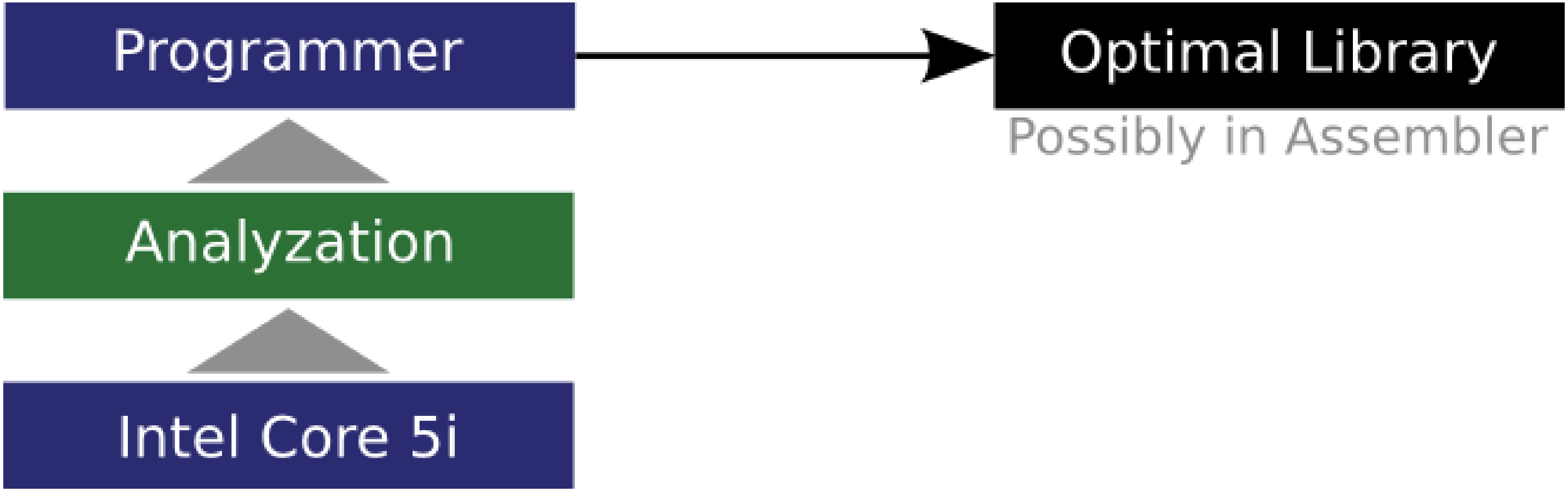
Traditional Approach

Hand-optimized libraries



Traditional Approach

Hand-optimized libraries



Programmer

Analyzation

Intel Core 5i

Optimal Library

Possibly in Assembler

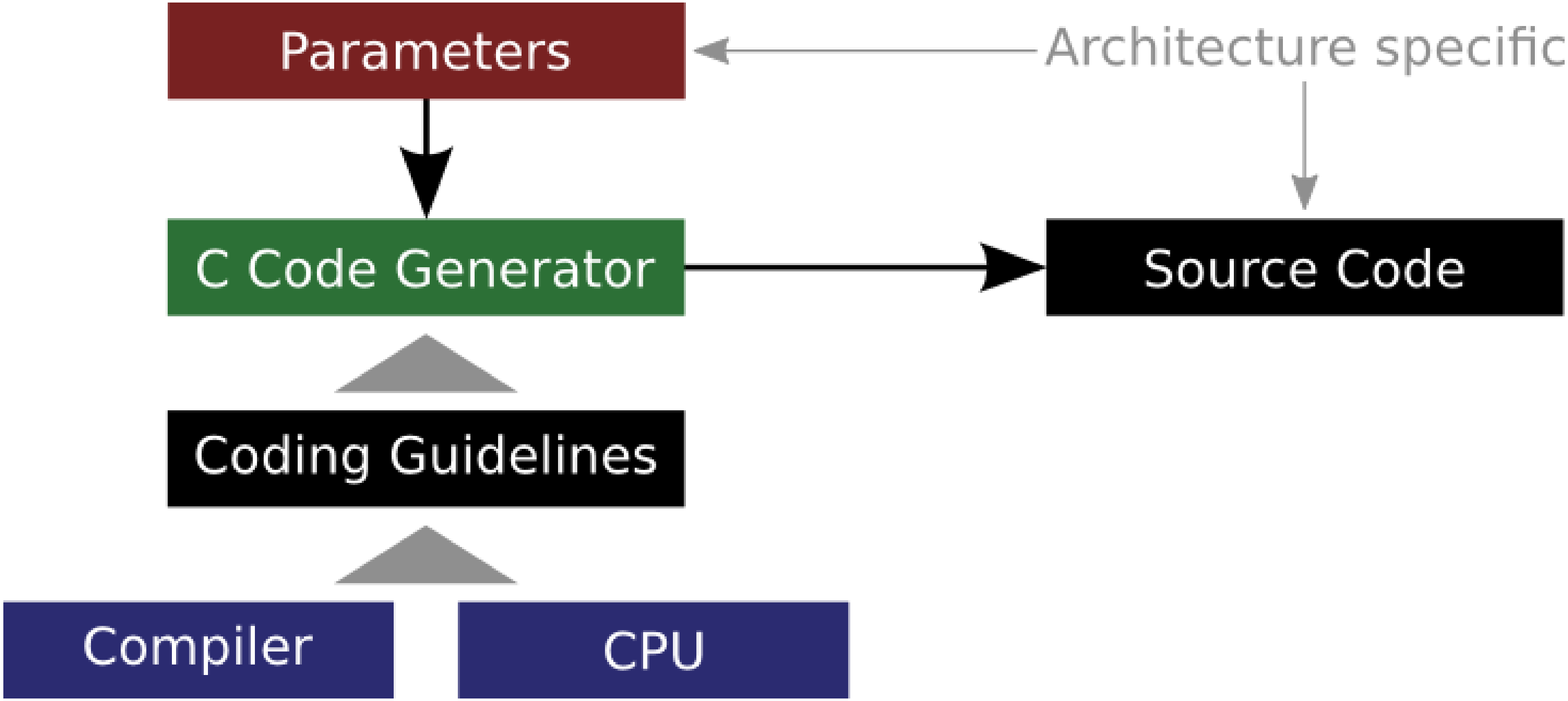
Traditional Approach

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



Parameters

Architecture specific

C Code Generator

Source Code

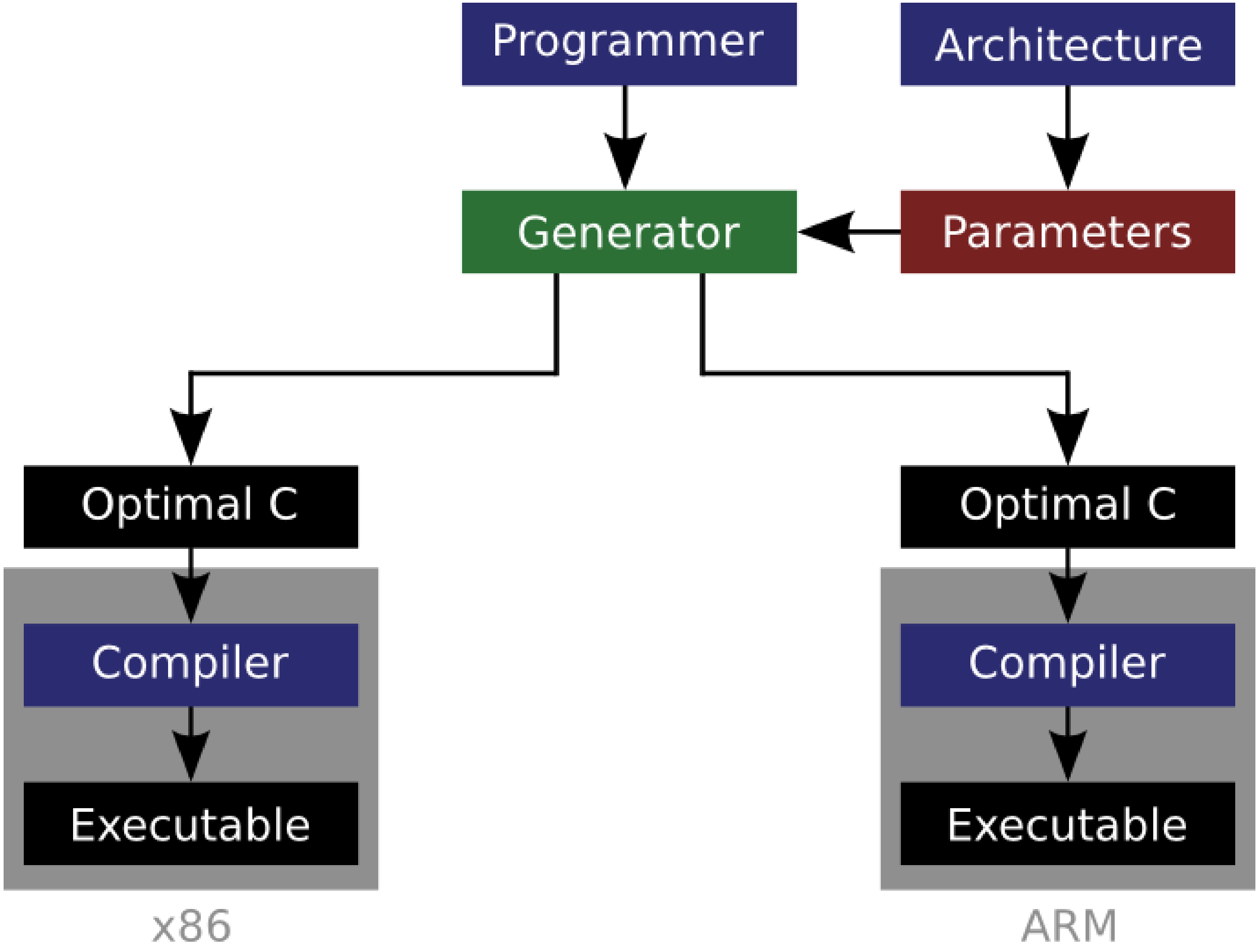
Coding Guidelines

Compiler

CPU

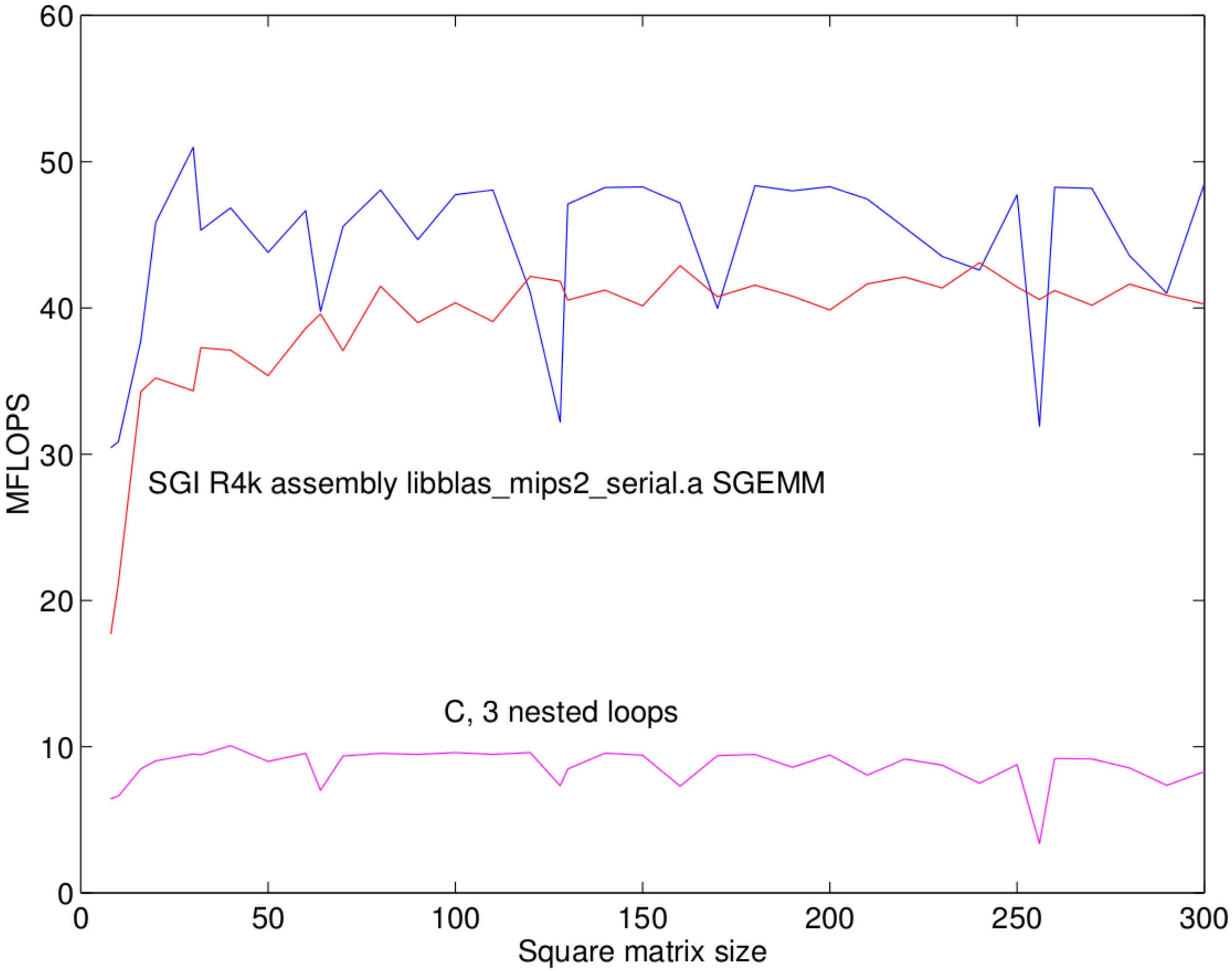
PHiPAC Approach

Parameters are architecture specific



PHiPAC Approach

Look ahead



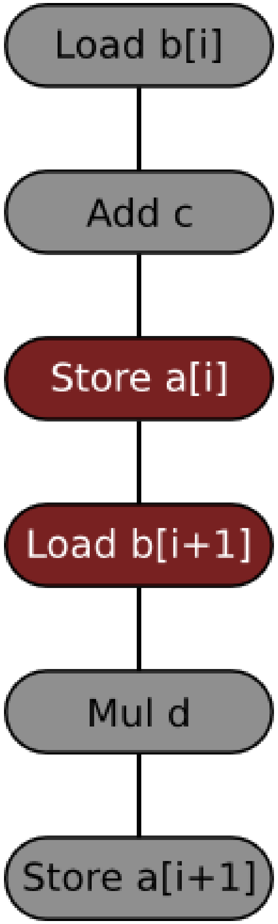
Source: PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology

Coding Guidelines

Remove false dependencies

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

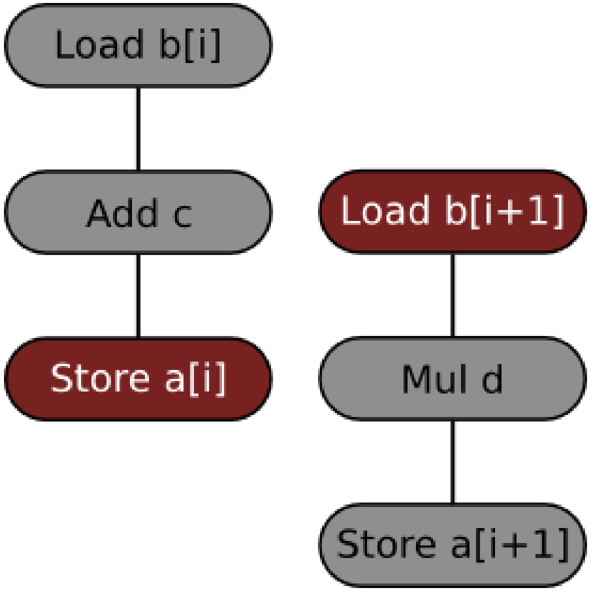
$$\texttt{\&a[i]} \overset{?}{==} \texttt{\&b[i+1]}$$



Coding Guidelines

Remove false dependencies

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



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

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

Coding Guidelines

Scalar Replacement: Exploit Register File

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

```
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

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

```
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)

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

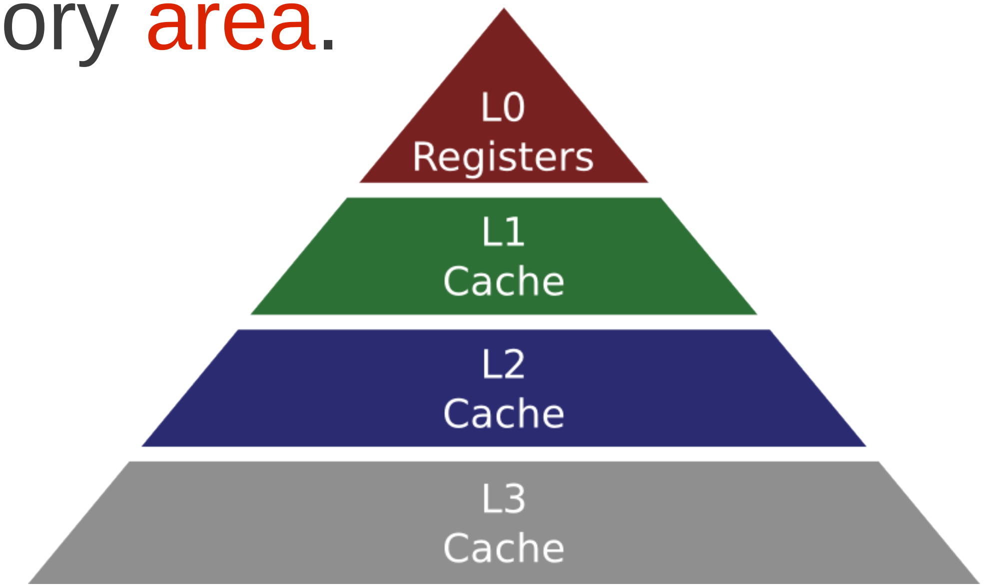
```
movl   (%ecx), %eax
movl 16(%ecx), %ebx
movl 32(%ecx), %edx
movl 48(%ecx), %esi
```

(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

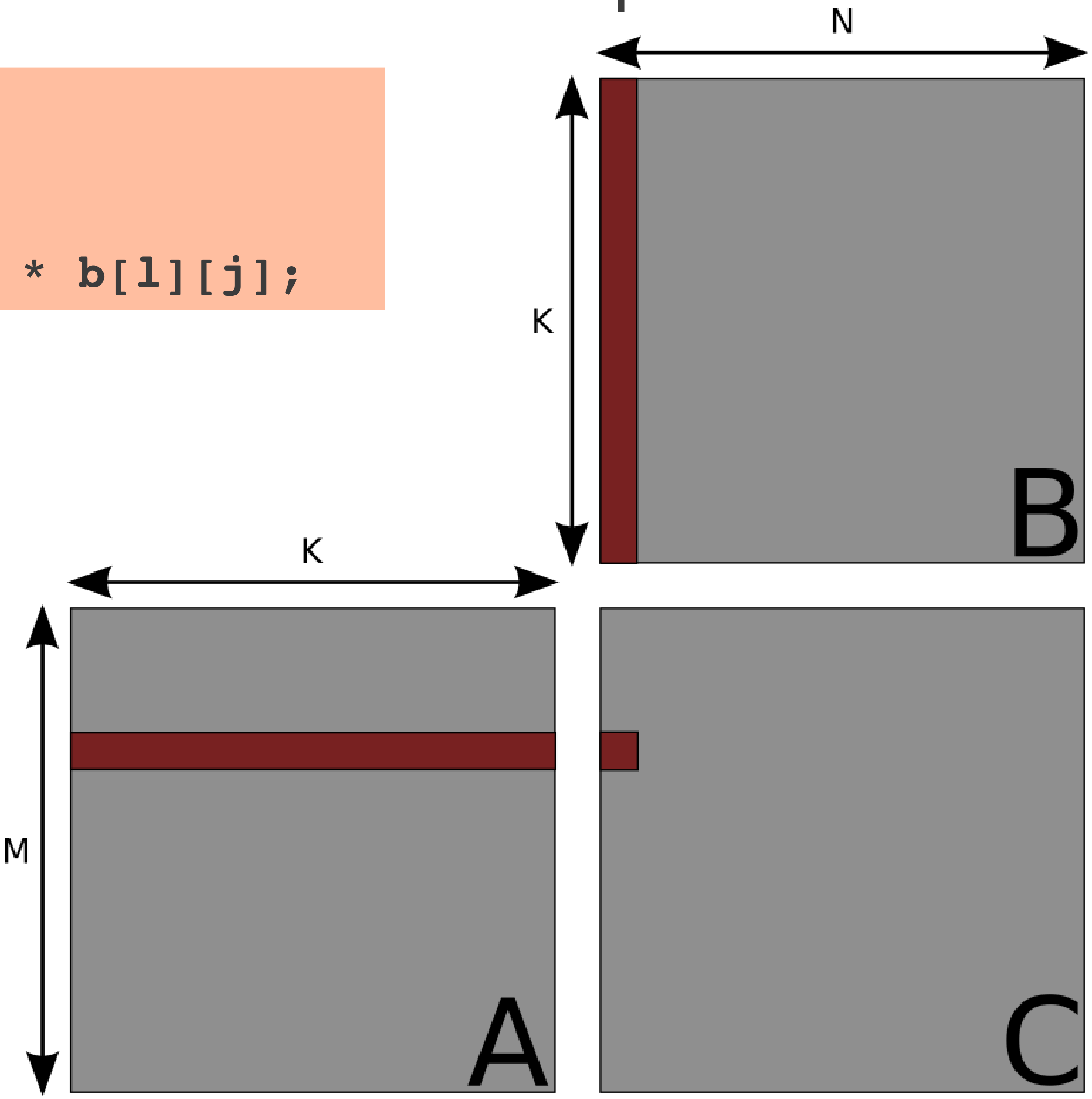
Summary

Guideline	Effect	Parameterizable
Use Scalar Replacement to remove false dependencies	Parallel execute of independent operations	
Use Scalar Replacement exploit register file	Decreased memory bandwidth	**yes**
Use Scalar Replacement minimize pointer updates	Compressed instruction sequence	
Hide multiple instruction FPU latency	Independent execution of instructions in pipelined CPUs	
Balance the instruction mix	Increased instruction throughput	
Increase locality	Increased cache performance	**yes**
Minimize branches	Decrease number of pipeline flushes	
Loop unrolling	Compressed instruction sequence	**yes**
Convert integer multiplies to adds	Decrease instruction latency	

Matrix Multiplications

Simplest Approach: Three nested loops

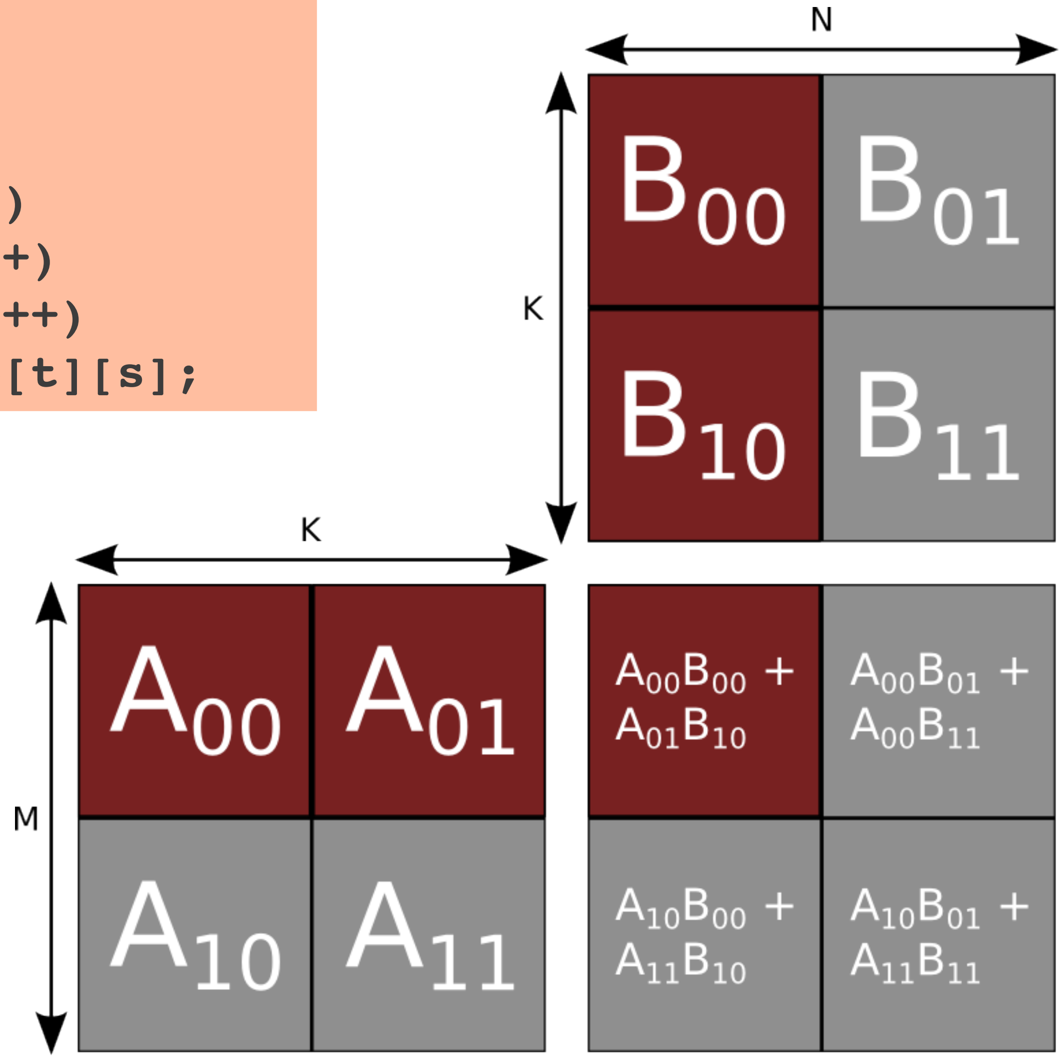
```
for (i=0; i<M; i++)
 for (j=0; j<N; j++)
  for (l=0; l<K; l++)
   c[i][j] += a[i][l] * b[l][j];
```



Block Matrix Multiplication

General Approach

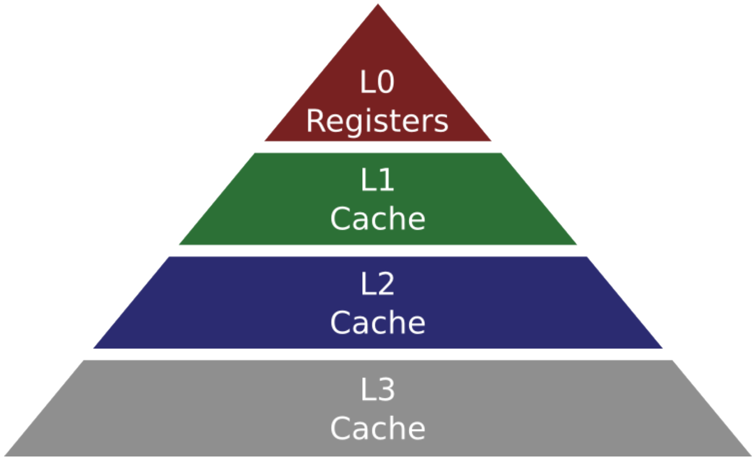
```
for (i=0; i<M; i+=MBlock)
 for (j=0; j<N; j+=NBlock)
  for (l=0; l<K; l+=KBlock)
   for (r=i; r<i+MBlock; r++)
    for (s=i; s<i+NBlock; s++)
     for (t=i; t<i+KBlock; t++)
      c[r][s] += a[r][t] * b[t][s];
```

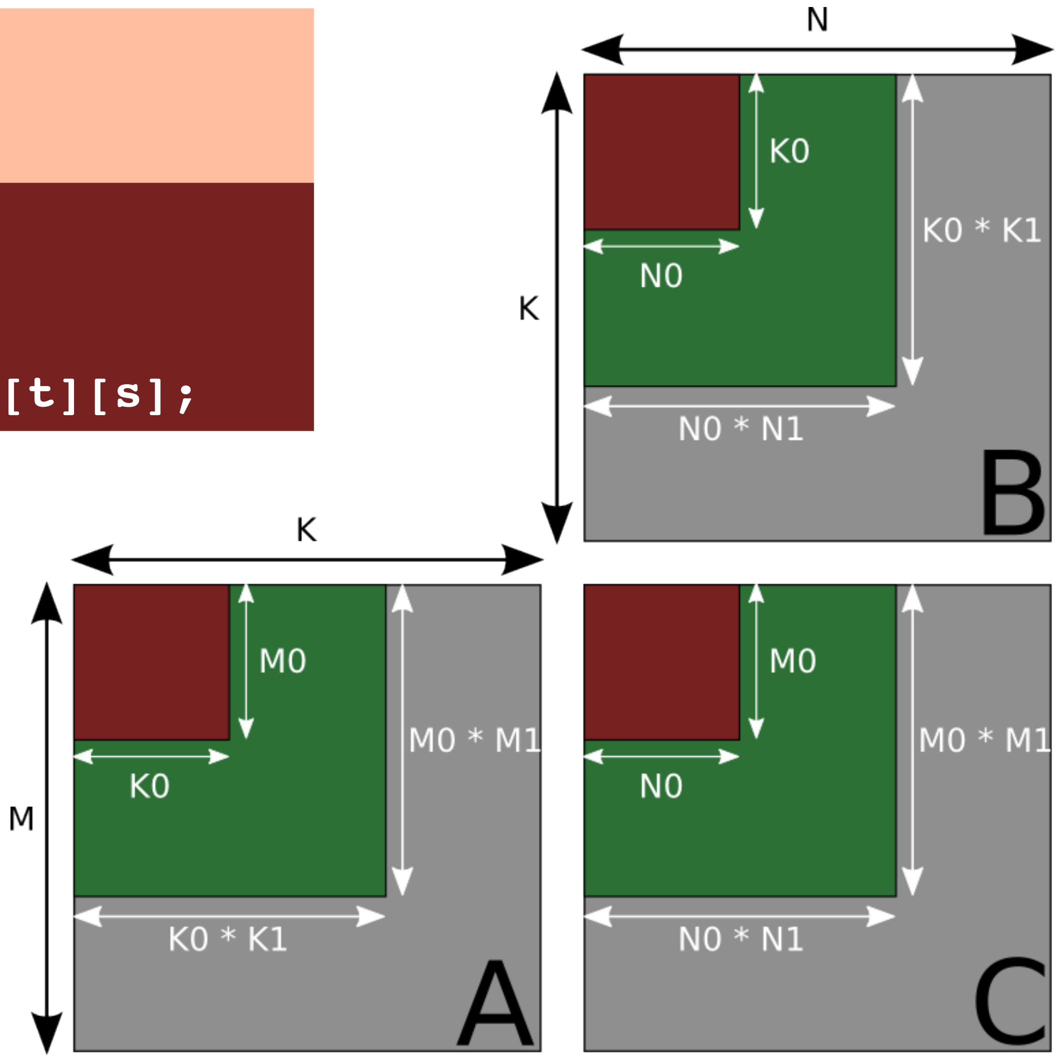


Matrix Multiplications

Choose appropriate block sizes

```
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
```

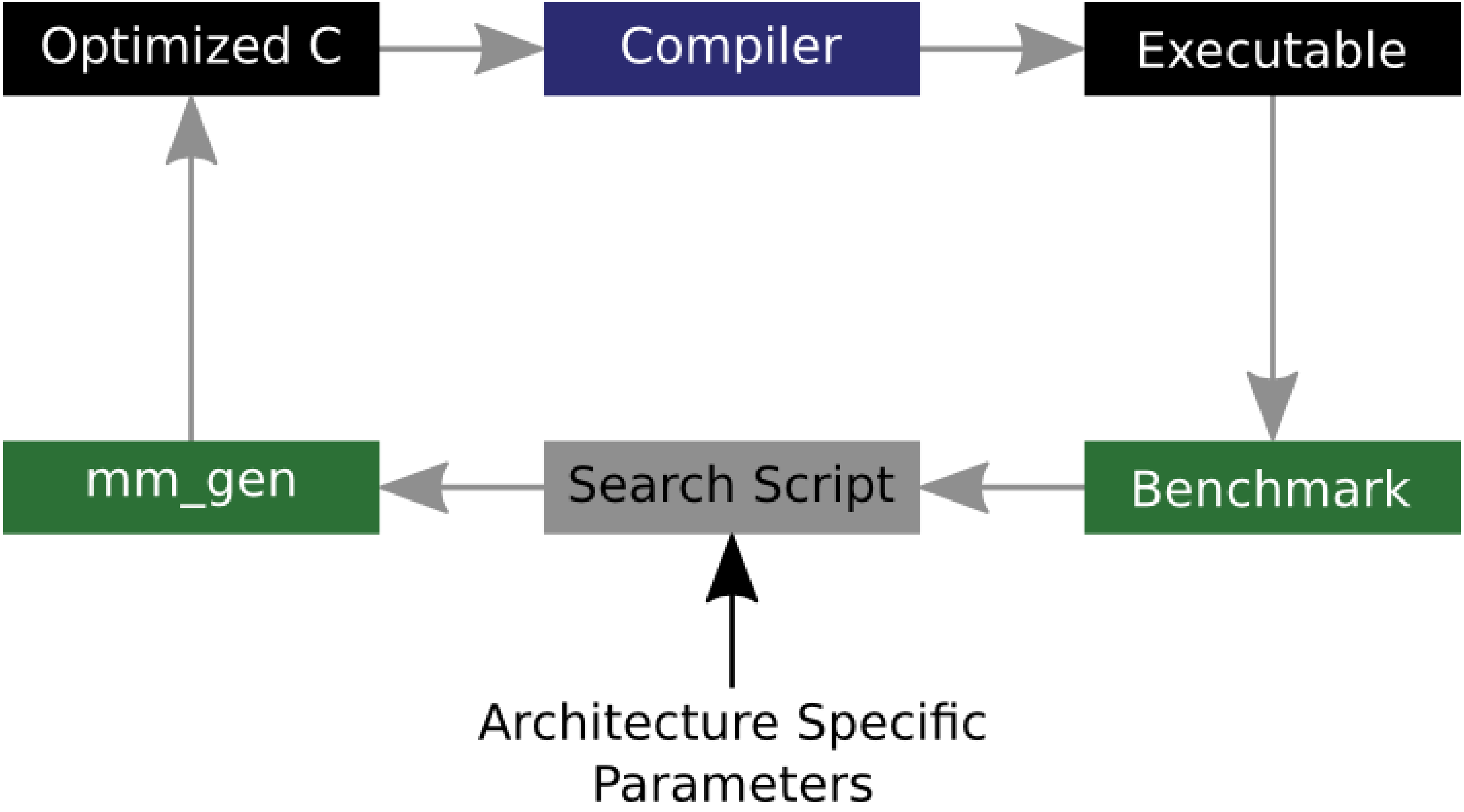
```
do { /* */
  do { /*...*/
    do { /*...*/
      _b0 = bp[0]; _b1 = bp[1];
      bp += Bstride;
      _a0 = ap_0[0];
      c0_0 += _a0*_b0; c0_1 += _a0*_b1;
      _a1 = ap_1[0];
      c1_0 += _a1*_b0; c1_1 += _a1*_b1;

      _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

Benchmark

Search Script

mm_gen

Architecture Specific
Parameters

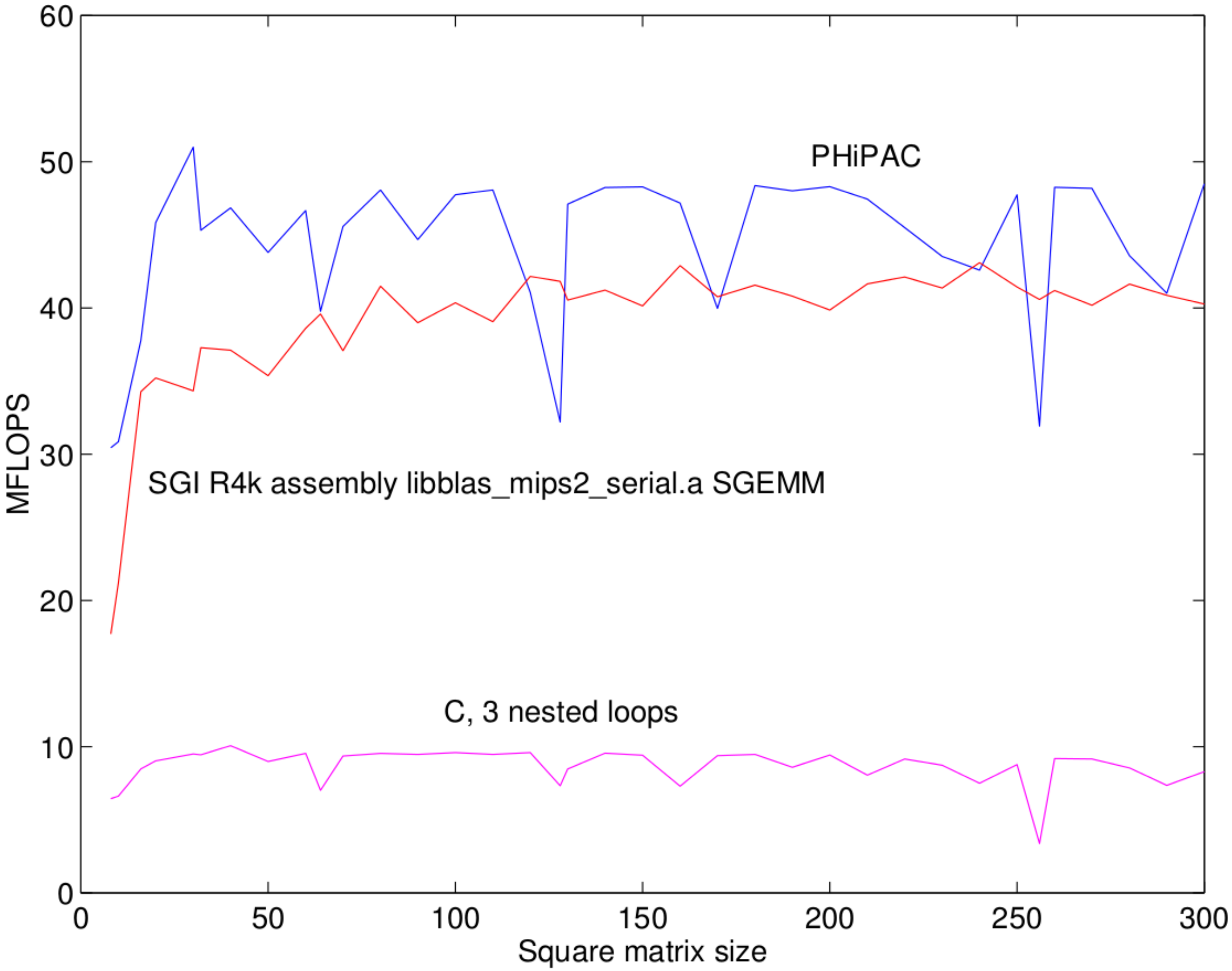
Finding Optimal Block Sizes

Example: Finding the L1 Parameters

- We have to limit the parameter space
- For the square case DxD
- We search the neighborhood centered at $3\mathrm{D}^2 = L_1$
- We set $M_{1,} K_{1,} N_1$ to the values $\phi D / M_0$
- Where $\phi \in (0.25, 0.5, 1.0, 1.5, 2.0)$
- => 125 Combinations

Results

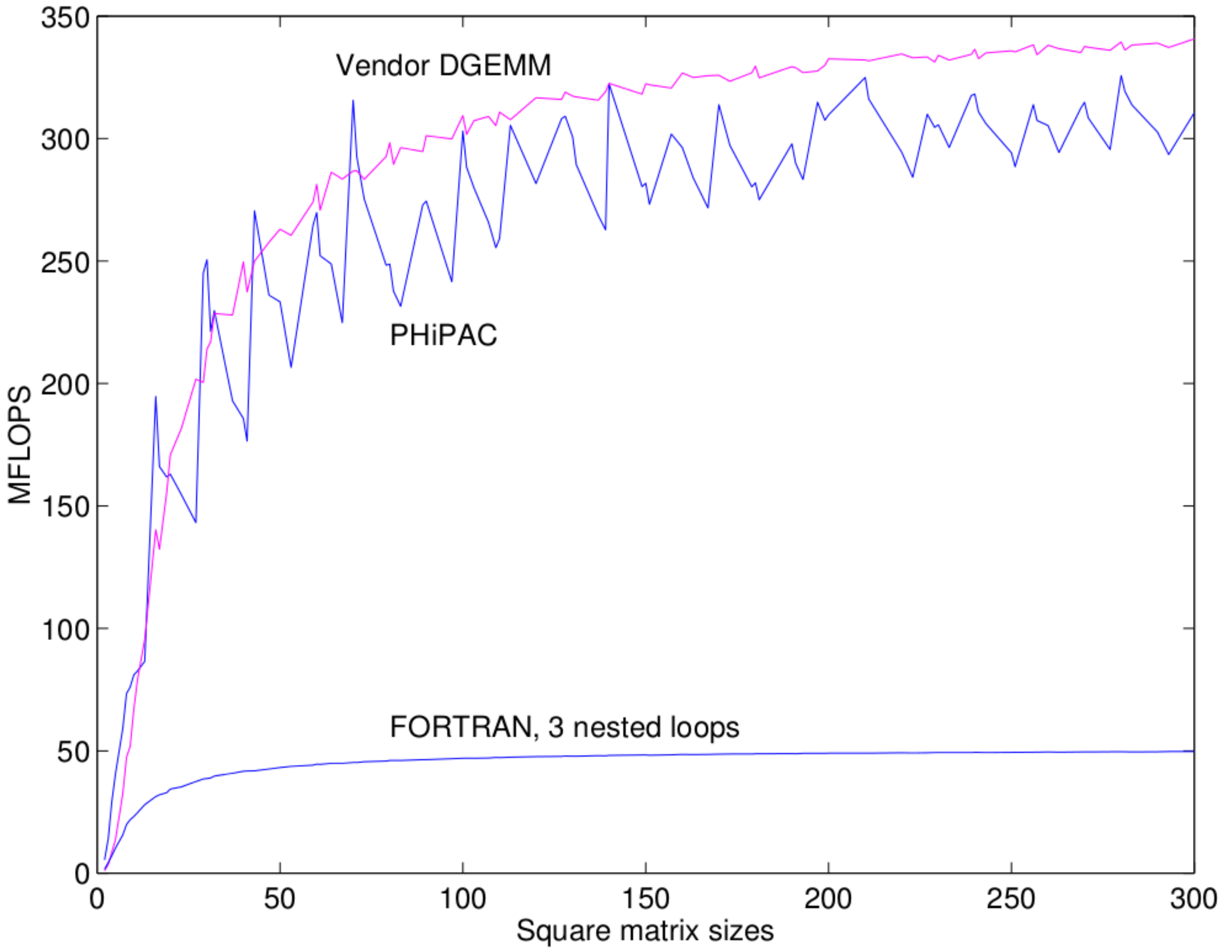
Example (Single Precision Matrix Mult. on a 100MHz SGI Indigo R4K)



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/phipac/