Table of Contents

1. Platform Architecture.................................................................................................................. 2
  1.1. CPU Information.................................................................................................................. 2
  1.2. Memory Information............................................................................................................. 4
    1.2.1. Using /proc/cpuinfo.................................................................................................... 4
    1.2.2. Reading from /sys/devices/system/cpu/cpu0/cache/..................................................... 5
    1.2.3. Pulling Memory Metrics from PAPI............................................................................... 6
  1.3. Operating System Information............................................................................................ 7
    1.3.1. Query uname -a............................................................................................................. 7
    1.3.2. Query lsb_release -idrc.............................................................................................. 8
  1.4. gcc Information................................................................................................................... 8
2. Timing and Performance Results............................................................................................... 9
  2.1. Numerical Recipes Code...................................................................................................... 9
    2.1.1. matrix_prod() with various gcc optimizations.......................................................... 9
    2.1.2. matrix_prod() with different for loop orders (no gcc optimization).......................... 12
    2.1.3. Optimization through blocking the matrix multiply operation..................................... 15
  2.2. ATLAS Code...................................................................................................................... 18
1. Platform Architecture

1.1. CPU Information

Obtaining CPU information in Linux is simple and straightforward. Most, if not all, relevant information may be obtained from the proc mount on /proc in the cpuinfo file.

Number of CPUs:

$ cat /proc/cpuinfo | grep processor
processor : 0
processor : 1

We can see that the system has two CPUs.

CPU Model

$ cat /proc/cpuinfo | grep "model name"
model name : AMD Opteron(tm) Processor 244
model name : AMD Opteron(tm) Processor 244

We can see that both CPUs are AMD Opterons.

CPU Clock

$ cat /proc/cpuinfo | grep MHz
cpu MHz : 1809.272
cpu MHz : 1809.272

Both CPUs are clocked at 1.8Ghz

Do we have an FPU?

$ cat /proc/cpuinfo | grep fpu
fpu : yes

Yes.
Bells, whistles, and extended instruction sets.

$ cat /proc/cpuinfo | grep flags

flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
fxsr sse sse2 syscall nx mmxext lm 3dnowext 3dnow rep_good

We can see, again, that the processors have an FPU. We can also see from the \texttt{lm} flag that the CPUs are both 64-bit. For an explanation of the remaining flags is it best to query the header file \texttt{cpufeature.h} from within the kernel source tree.

$ cat /usr/src/linux/include/asm-i386/cpufeature.h | grep FEATURE

\begin{verbatim}
#define CPU_FEATURE_P(CAP, FEATURE) test_bit(CAP, X86_FEATURE_##FEATURE##_BIT)
#define X86_FEATURE_FPU        (0*32+ 0) /* Onboard FPU */
#define X86_FEATURE_VME        (0*32+ 1) /* Virtual Mode Extensions */
#define X86_FEATURE_DE         (0*32+ 2) /* Debugging Extensions */
#define X86_FEATURE_PSE        (0*32+ 3) /* Page Size Extensions */
#define X86_FEATURE_TSC        (0*32+ 4) /* Time Stamp Counter */
#define X86_FEATURE_MSR        (0*32+ 5) /* Model-Specific Registers, RDMSR, WRMSR */
#define X86_FEATURE_PAE        (0*32+ 6) /* Physical Address Extensions */
#define X86_FEATURE_MCE        (0*32+ 7) /* Machine Check Architecture */
#define X86_FEATURE_CX8        (0*32+ 8) /* CMPXCHG8 instruction */
#define X86_FEATURE_APIC       (0*32+ 9) /* Onboard APIC */
#define X86_FEATURE_SEP        (0*32+11) /* SYSENTER/SYSEXIT */
#define X86_FEATURE_MTRR       (0*32+12) /* Memory Type Range Registers */
#define X86_FEATURE_PGE        (0*32+13) /* Page Global Enable */
#define X86_FEATURE_MCA        (0*32+14) /* Machine Check Architecture */
#define X86_FEATURE_CMOV       (0*32+15) /* CMOV instruction (FCMOVCC and FCOMI too if FPU present) */
#define X86_FEATURE_PAT        (0*32+16) /* Page Attribute Table */
#define X86_FEATURE_PSE36      (0*32+17) /* 36-bit PSEs */
#define X86_FEATURE_PN         (0*32+18) /* Processor serial number */
#define X86_FEATURE_CLFLSH     (0*32+19) /* Supports the CLFLUSH instruction */
#define X86_FEATURE_DTES       (0*32+21) /* Debug Trace Store */
#define X86_FEATURE_ACPI       (0*32+22) /* ACPI via MSR */
#define X86_FEATURE_MMX        (0*32+23) /* Multimedia Extensions */
#define X86_FEATURE_FXSR       (0*32+24) /* FXSAVE and FXRSTOR instructions (fast save and restore */
#define X86_FEATURE_XMM        (0*32+25) /* Streaming SIMD Extensions */
\end{verbatim}
1.2. Memory Information

The proc filesystem also houses some information about the system memory. Let's begin, again, with /proc/cpuinfo to find some basic information about the system's cache.

1.2.1. Using /proc/cpuinfo

```
$ cat /proc/cpuinfo | grep -e TLB -e cache -e address
cache size : 1024 KB
TLB size : 1024 4K pages
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
```

Here several metrics are reported, but there is actually a fair amount of debate from within the Linux community regarding the output we are looking at here. The cache information delivered by cpuinfo is found by many to be unsuitable since it doesn't specify which cache (or combination of caches) it is talking about. Furthermore, on machines utilizing hyperthreading technology, the cache reported here can be misleading since hyperthreading reserves about half your cache in order to provide two virtual processors. The TLB size
is the size of the translation lookaside buffer, which is a CPU cache that is used to optimize the process of virtual address translation.

### 1.2.2. Reading from /sys/devices/system/cpu/cpu0/cache/

While /proc/cpuinfo is a pretty good place to peek first to get a good idea of what type of system you are using, the /sys branch of the filesystem is better queried to get detailed information about the system cache. Here is a very simple bash script I quickly wrote to display some more detailed information about the system cache pulled from /sys

```bash
#!/bin/bash

echo System Cache Information for tux64

echo by James Shackleford 10-20-08

echo

for j in $(seq 0 2); do
    echo L`cat /sys/devices/system/cpu/cpu0/cache/index$j/level`
    echo `cat /sys/devices/system/cpu/cpu0/cache/index$j/type` Cache
    echo Total Size : `cat /sys/devices/system/cpu/cpu0/cache/index$j/size`
    echo Line Size : `cat /sys/devices/system/cpu/cpu0/cache/index$j/coherency_line_size`
    echo Number of lines: $((`cat /sys/devices/system/cpu/cpu0/cache/index$j/number_of_sets` * `cat /sys/devices/system/cpu/cpu0/cache/index$j/ways_of_associativity`))
    echo Associativity : `cat /sys/devices/system/cpu/cpu0/cache/index$j/ways_of_associativity`
    echo
done;
```

Which produces the following output on tux
Output of script: scripts/cacheinfo.sh

System Cache Information for tux64
by James Shackleford 10-20-08

L1 Data Cache
Total Size : 64K
Line Size : 64
Number of lines: 1024
Associativity: 2

L1 Instruction Cache
Total Size : 64K
Line Size : 64
Number of lines: 1024
Associativity: 2

L2 Unified Cache
Total Size : 1024K
Line Size : 64
Number of lines: 16384
Associativity: 16

1.2.3. Pulling Memory Metrics from PAPI

Obtaining information about the system cache via PAPI's included memory information binary is simple:

$ /usr/local/papi/bin/papi_mem_info
Test case: Memory Information.

L1 Instruction TLB: Number of Entries: 512; Associativity: 4
L1 Data TLB: Number of Entries: 512; Associativity: 4

L1 Instruction Cache:
  Total size: 64KB
  Line size: 64B
  Number of Lines: 1024
  Associativity: 2

L1 Data Cache:
The information provided by PAPI agrees with the cache information provided from /sys and extends upon the TLB information provided by /proc/cpuinfo. (cpuinfo declared 1024 4KB pages, while PAPI shows how the TLB cache is evenly split between data and instruction entries).

1.3. Operating System Information

Several utilities that are standard across all Linux distributions will provide us with all relevant information about the OS.

1.3.1. Query uname -a

The uname utility provides the following system information, in the following order when triggered with the flag -a (with the results for our system in parenthesis)

- kernel name (Linux)
- hostname (tux64-03.cs.drexel.edu)
- kernel release (2.6.24.3-nfsgroups-perfctr)
- kernel version (#1 SMP Thu Jul 31 16:34:17 EDT 2008)
- name of hardware platform (x86_64)
- operating system (GNU/Linux)

$ uname -a
Linux tux64-03.cs.drexel.edu 2.6.24.3-nfsgroups-perfctr #1 SMP Thu Jul 31 16:34:17 EDT 2008 x86_64
1.3.2. Query lsb_release -idrc

The acronym LSB stands for Linux Standard Base and the command lsb_release prints distribution specific information to stdio. Every Linux distribution should provide the lsb_release facility. The following flags are used:

- -i Distributor ID (Ubuntu)
- -d Description (Ubuntu 8.04.1)
- -r Release Number (8.04)
- -c Codename (hardy)

```bash
$ lsb_release -idrc
Distributor ID: Ubuntu
Description: Ubuntu 8.04.1
Release: 8.04
Codename: hardy
```

1.4. gcc Information

Information regarding gcc is easily obtainable:

```bash
jas64@tux64-01:/A1/bin$ gcc --version
gcc (GCC) 4.2.4 (Ubuntu 4.2.4-1ubuntu3)
Copyright (C) 2007 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```
2. Timing and Performance Results

2.1. Numerical Recipes Code

Here we will test and collect figures of merit from the matrix multiplication function provided by Numerical Recipes in C. We will first experiment with the unoptimized version, and then we will see how level 2 and level 3 optimization provided by gcc affects the algorithm's performance. Following this, we will see how changing the order of the nested for loop structure used to implement the algorithm will affect performance.

2.1.1. matrix_prod() with various gcc optimizations

So, let us take a look at the performance benchmarks for matrix_prod() included in Numerical Recipes in C. Here I will show the performance benchmarks with various levels of optimization provided by gcc. Note the legend in the top right of each graph denoting the optimization level of a given data set.

Figure 1  MFLOPS vs Matrix Size [n x n] shown for various optimization offered by gcc.
Figure 1 shows the MFLOPS for increasing array sizes for different optimization flags used in conjunction with the standard Numerical Recipes in C ijk ordered matrix_prod() matrix multiply function. As can be seen from the figure, we don't really gain that much by utilizing gcc's optimization flags. The general character of the data remains about the same; we gain a few MFLOPS here and there for different optimization flags for different matrix sizes. It is apparent here that the limitation here is the algorithm itself and not necessarily gcc's transcription to machine code. If anything, you could argue that –O3 is the way to go since it seems to ride the top of the performance curve for every value of n tested, but, again, using –O3 only gains us some spare change in performance. We can do better by modifying the algorithm itself. Figure 2, below, shows the L1 data cache misses per FLOP for the same data set. We can see that the L1 data cache misses are the same for every optimization flag so optimization is not helping our locality.

Figure 2  L1 Data Cache Misses vs Matrix Size [n x n] shown for various optimization levels offered by gcc.
Finally, before we move on to modifying the algorithm, let's take a quick look at the total number of instructions executed per FLOP for any given array size during the course of a matrix multiplication. For all all levels of optimization we asymptotically approach 23 instructions per FLOP as we increase the array size. We can therefore infer that the core of the algorithm requires approximately 23 instructions to function. There may be some startup and cleanup code that manifests itself in the graph for smaller matrix multiplications. Regardless, optimization flags are not going to help us here; so let's move on to optimizing the matrix multiplication algorithm itself.

![Graph of Total Instructions per FLOP vs Matrix Size](image)

**Figure 3** Total Instructions per FLOP vs Matrix Size \([n \times n]\) shown for various optimization levels offered by gcc.
2.1.2. matrix Prod() with different for loop orders (no gcc optimization)

![MFLOPS vs Matrix Size [n x n] shown for various for-loop orderings.](image)

What we have done is here is modify the order of the triply nested for-loops that constitute the matrix_prod() matrix multiplication algorithm. We can see from Figure 4 that for large matrices we can just about double our MFLOPS by properly constructing the nested for-loop structure to take advantage of spatial locality to some degree. By forming the nested for-loop structure to work optimally with the row-major nature of C array storage in memory, we can eliminate unnecessary memory strides and increase the algorithm's performance by decreasing L1 data cache misses. Again, referring to Figure 4, we can see that ijk and kij algorithms performed the best, which is accomplished by minimizing L1 data cache misses. Referring to Figure 5, we can empirically see that this analysis is correct. However, we can further increase both the spatial and temporal locality of the algorithm by implementing block matrix multiplication to keep the working set adequately sized for the L1 cache.
Figure 5   L1 Data Cache Misses vs Matrix Size [n x n] shown for various for-loop orderings.
Figure 6  Total Instructions per FLOP vs Matrix Size [n x n] shown for various for-loop orderings.
2.1.3. Optimization through blocking the matrix multiply operation

As shown in Figure 7, implementing the block matrix multiplication increases the efficiency of the algorithm. By analyzing Figure 8, we can see that the L1 data cache misses have been dramatically reduced by implementing the blocking. We can see that the algorithm's performance increases when the block size is increased from five to twenty-five. The temporal and spacial locality introduced by the blocking algorithm noticeably improves performance; we can attribute this directly to the deceased rate of L1 data cache misses.

Figure 7  MFLOPS vs Matrix Size [n x n] shown for different block sizes. The unblocked ijk algorithm is also shown for reference and comparison.
Figure 8  L1 Data Cache Misses per FLOP vs Matrix Size [n x n] shown for different block sizes.

The unblocked ijk algorithm is also shown for reference and comparison.
Furthermore, we can see the blocking algorithm utilizes fewer instructions per FLOP for a block size of twenty-five. However, interestingly, we can also see that for a block size of five, the instructions executed per FLOP actually increase by half an instruction per FLOP over the standard ijk ordered Numerical Recipes algorithm. Despite this, the blocked algorithm utilizing a block size of five outperforms the ijk ordered algorithm by virtue of invoking fewer L1 data cache misses, which just goes to show that executed instructions per FLOP is by no means an adequate benchmark of how efficient a given algorithm is.

![Figure 9](image-url)  
**Figure 9**  Total Number of Instructions per FLOP vs Matrix Size [n x n] shown for different block sizes. The unblocked ijk algorithm is also shown for reference and comparison.
2.2. ATLAS Code

Finally, here we utilize the cblas_sgemm() function included in ATLAS 3.8.1 to implement the matrix multiplication. We can plainly see that ATLAS outperforms all other previously mentioned methods of matrix multiplication by about two orders of magnitude. Interestingly, the L1 data cache utilization of the blocked ijk multiply with block size of twenty-five is about on par with that of ATLAS’s implementation of cblas_sgemm() with both algorithms marking about 0.005 L1 data cache misses per FLOP. However, as we can see from Figure 12, the instructions executed per FLOP for the cblas_sgemm() function is around 2 to the blocking algorithm's 21—a reduction of one whole order of magnitude. The machine specific tuning of ATLAS combined with its superior algorithms results in all other algorithms tested here failing in comparison.
Figure 11  L1 Data Cache Misses per FLOP vs Matrix Size [n x n] shown for different block sizes.
The unblocked ijk and blocked ijk (b=25) algorithms are also shown for reference and comparison.
Figure 12  Total Instructions per FLOP vs Matrix Size [n x n] shown for different block sizes. The unblocked ijk and blocked ijk (b=25) algorithms are also shown for reference and comparison.