SANDIA REPORT
SAND2017-1255
Unlimited Release
Printed February 2, 2017

PAPI Recipes for Sandy Bridge Systems

Douglas M. Pase

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: http://www.ntis.gov/search

SAND2017-1255
Printed February 2, 2017
Unlimited Release

PAPI Recipes for Sandy Bridge Systems

Douglas M. Pase
9326 Scientific Apps & User Support
Sandia National Laboratories
P. O. Box 5800
Albuquerque, New Mexico 87185-MS 0807

Abstract

This report examines the PAPI interface to the Intel Performance Monitor Counters (PMC). The PMC is a hardware mechanism for reporting performance-related processor core and memory events. These events include such things as an L1 cache miss or a load instruction completed. Information such as this can be extremely useful when working to improve application performance.

In this report we explore various uses of the counters for examining application performance. We provide a useful selection of event types to count for general use. We provide definitions of the counters as they are defined on Intel Sandy Bridge. We also give definitions of commonly useful derived metrics, such as CPI, that highlight performance issues. We show how to use them and give several examples of their use.

This approach also extends to systems based on Intel Broadwell processors as well as others.

TABLE OF CONTENTS

FIGURES

TABLES

NOMENCLATURE

Abbreviation	Definition
ALU	Arithmetic Logic Unit
AVX	Advanced Vector Extensions
BLAS	Basic Linear Algebra Subroutine package mathematical library
CPI	Clocks per instruction
CPU	Central Processing Unit
DAXPY	Double-precision vector A (times) X plus Y routine
DGEMM	Double-precision general matrix multiplication routine
DGETRF	Double-precision general matrix factorization with partial pivoting
DP	Double-precision
E5-2670	2.6 GHz Intel E5-2670 Sandy Bridge processor
FLOP	Floating-point operation
FP	Floating-point
GB/s	Gigabytes per second, or 1,000,000,000 bytes per second
GF/s	Gigaflops per second, billions of floating-point operations per second
GHz	Gigahertz, or 1,000,000,000 cycles per second
KiB	Kilobytes, or 1,024 bytes
L1, L2, L3	Level 1 (or 2 or 3) cache
LAPACK	Linear Algebra Package of mathematical subroutines
LU	Matrix factorization into lower- and upper-triangle form
MB	Megabytes, or 1,000,000 bytes
MB/s	Megabytes per second, or 1,000,000 bytes per second
MFLOP/sec	Megaflops per second, or 1,000,000 DP floating-point operations per second
MHz	Megahertz, or 1,000,000 cycles per second
MiB	Megabytes, or 1,024 x 1,024 bytes
MPI	Message Passing Interface
NUMA	Non-uniform memory access
OPS	Operations
PAPI	Performance Application Program Interface
`PAPI_*`	See Appendix A for definitions of PAPI events
PCHASE	Pointer chasing micro-benchmark that measures memory latency
PMC	Performance Monitor Counters
QPI	Quick Path Interconnect
TDP	Thermal Design Point

1. INTRODUCTION

Optimization is the process of maximizing benefit gained while minimizing the resources used. Application performance optimization attempts to provide the most efficient use of computer system resources in order to minimize the amount of time to complete a task. Successful optimization requires detailed knowledge about how operations within the application are making use of the available hardware resources, including low-level processor core components such as various levels of cache, floating-point units, and so forth.

Gathering this information requires multiple steps, including: selecting the instrumentation to be used, placing the instrumentation at key locations within the code, gathering the data from the running application, and analyzing the data for opportunities to improve the code.

The Intel Performance Monitor Counters, or PMC, is a set of eight flexible counters per processor core that track core-level events. Core-level events include the completion of any instruction, or the completion of only certain types of instructions, such as branch or floating-point instructions. They also include cache misses for each level of cache. PMC counters give a detailed view into what the processor is doing to execute an application, and that information can be extremely useful for code optimization.

PAPI is a library package that provides convenient access to the Intel PMC facility. It provides subroutines to set up, start, and stop the counters. It provides standardized names for each class of events, and handles the hardware specific event codes that often make using such facilities difficult. PAPI can be accessed through its native API, and it is used by other packages, such as TAU and Open|SpeedShop, to provide low-level performance data through higher-level performance tools.

But PMC and PAPI are complex and often subtle facilities. PAPI itself is fairly simple, but PMC is not, and though PAPI hides the more complex mechanics of PMC from the user, it cannot hide the complexities of interpreting the output of the counters themselves.

This report attempts to accomplish several things. First, we look carefully at what the counters are actually counting. We look for values that are reported consistently from one run to the next, and we try to establish what units are being counted. For example, does a certain event represent an instruction or an operation? Vector addition can be a single instruction that performs multiple floating-point operations. Which is being counted?

Second, we look for ways to add the counters to a program. TAU and Open|SpeedShop offer ways to use the counters without recompiling the application, but they may count more than is actually wanted. Adding PAPI instrumentation manually is more effort, but it can give more narrowly focused results.

Third, we look for ways of using the event counts to learn something useful about the performance of an application. To do that we use four micro-benchmarks that are already well understood, to learn what the counters can tell us. We look at the counters to see how they map to program behavior.

But counters by themselves don't tell us enough about program performance. They only tell us how many events have taken place, not whether there is opportunity for improvement. So finally, we look at some derived metrics that give useful information, not just data, about application performance.

2. PROFILES AND TRACES

There are many types of instrumentation one might use to analyze program behavior. At the highest level, there are two major styles to choose from: profiling and tracing. Profiles aggregate the data over time, whereas traces record ordered sequences of events. Profiles allow a user to answer such questions as, "How much time did I spend in this subroutine?" or, "How many times was that block of code executed?" Profiles generally record resource usage, whatever the notion of a resource might be. Resources might be CPU time, wall clock time, execution passes, floating-point and integer operations, cache hits or misses, memory loads and stores, and so forth. But profiles do not record any notion of event ordering or time stamps, so they cannot tell you whether this block of code was executed before or after some other block of code. For that, a trace is required.

Traces record ordered sequences of events. The events may be time stamped, but that isn't always the case. Events may also record resource usage at the time of the event, but that is also not required. The sequence need not even be a total ordering of events. A trace might indicate that event *B* occurred before event *A*, and event *C* also occurred before event *A*, yet give no indication whether event *B* occurred before or after event *C*. Time stamps help impose a total ordering on events, but distributed systems may also have distributed clocks, so even time stamps may not resolve the issue.

Comparing profiles and traces, profiles tend to be much smaller than traces, with data that is most useful in figuring out where the problems might be hiding. They give you a "big picture" view of the program. Traces often give you much more detail, but can be much larger and more intrusive as a result.

Optimization often needs to be focused on portions of code where the most time is spent. The greater the time spent in a block of code, the greater the potential benefit of optimizing that code. Amdahl's Law talks about limits to program speed improvements when a portion of the code can be parallelized. Specifically, speed-up can be approximated by looking at the amount of serial code, parallel code, and the degree of parallelization, using the equation $S_n = 1/((1-P) + P/n)$. In this equation, S_n is the speed-up with n processors and P is the parallelizable portion of the code. As n gets very large, the equation asymptotically approaches $1/(1-P)$ and that is all you can do.

The point of Amdahl's Law is that performance improvements are always limited by the portion of code not being improved. That principle applies to *all* code optimization, not just running code in parallel. If the section of code you're working on uses 10% of the total running time, the best your efforts could ever achieve would be 1/(1-0.1) or approximately 11% improvement, and that assumes you can optimize it away completely. The conclusion is, generally speaking, for best results, start with the most time consuming portions of code. That is where you have the greatest potential improvement. And, to find out how much time (or other resource) each portion of code uses, start with a profile.

This document is intended to discuss PAPI rather than performance tool instrumentation in general, so from this point forward we assume the use of PAPI in the context of gathering profile data rather than traces. PAPI can certainly be used in generating trace data, though, and the principles discussed here are every bit as relevant to traces as they are to profiles.

3. SYSTEM ARCHITECTURE AND PAPI EVENT COUNTERS

OK, we're going to profile our code, maybe the whole program, or maybe only some small portion of it. What are we looking for? We want to focus our efforts on some of the more time consuming portions of code, sure. But it's not enough to know that this function or that code block uses 90% of the time. Besides, we don't need PAPI for that kind of information, anyway. PAPI can tell us that, but other tools, such as `gprof`, can tell us that kind of information more easily. We also need to have some idea of what the code is doing with that time, in order to have some idea of *how*, or even *whether*, we might try to improve it. For that we need to have at least a basic understanding of the underlying architecture.

A compute cluster consists of a collection of compute servers connected together by some sort of communication fabric, such as Infiniband or Ethernet. Compute servers are sometimes referred to as "compute nodes", and compute nodes communicate using a communication library such as MPI. The cluster may also have storage nodes, administrative nodes, log-in nodes, visualization nodes, etc. But for running a program, it is the compute nodes that are most important.

Each compute node consists of one or more processors, memory, storage devices and communication adapters. Typical compute nodes have two processors. Storage devices and communication adapters are attached through an I/O bridge integrated into one of the processors. Each processor has its own memory controller and memory, but both processors can use all memory. Each processor also has several cores, and each core has its own collection of Arithmetic Logic Units (ALUs) that do the work of executing its own stream of instructions.

Each core comes with its own 32 KiB L1 data cache, 32 KiB L1 instruction cache, 256 KiB L2 unified data and instruction cache, and 2.5 MiB shared L3 cache. L1 and L2 caches are only used by their respective core, but all L3 cache blocks are shared over an internal network. Memory references that miss L1 cache are forwarded to L2, references that miss L2 cache are forwarded to L3, and references that miss all blocks of L3 cache are passed on to the correct memory controller. References to local memory are passed to the local memory controller, while references to remote memory are passed over the QPI subsystem to be handled by the remote processor's memory controller.

A simplified block diagram of a processor can be seen in Figure 1.

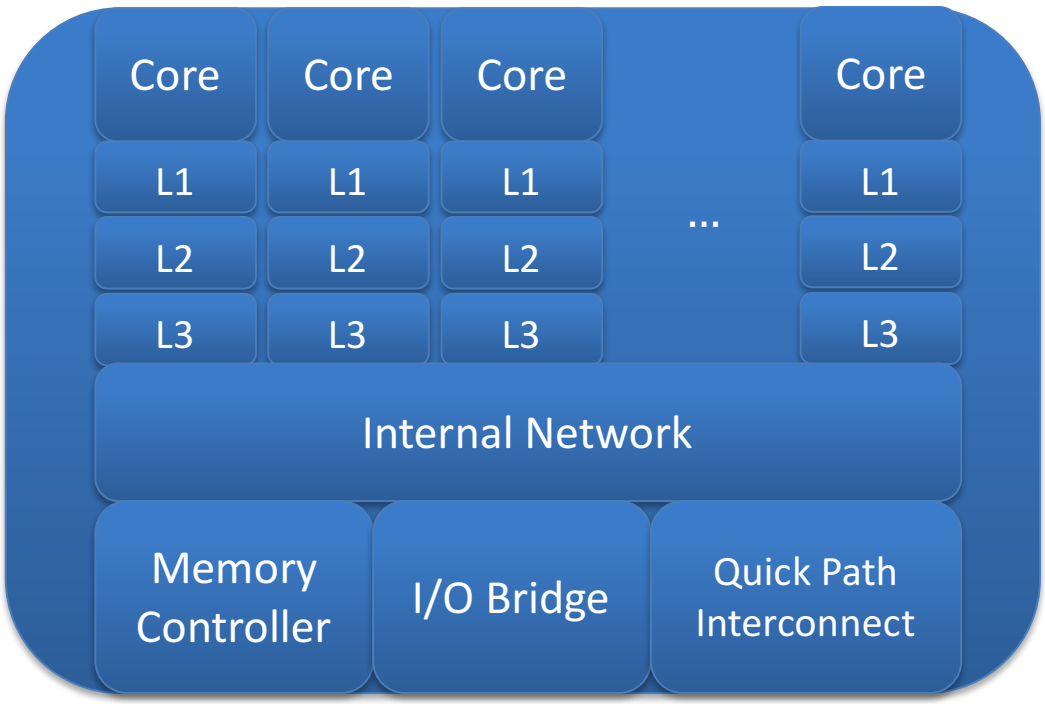

Figure 1 – Basic Processor Architecture

PAPI event counters focus almost exclusively on the execution of instructions within each core. Each core has a collection of hardware event counters that PAPI uses to gather data. Hardware events include things like the number of load and store instructions, floating-point instructions, or branch instructions that were executed, the number of references to each level of cache that missed, the total number of clock cycles completed, etc. A complete list of PAPI counters that are available on Sandy Bridge systems, along with their description, is given in Appendix A. PAPI counters do *not* include events related to the operation of storage or communication devices, or to the execution of operating system services. Neither does it count events related to, or distinguish between local *vs.* remote memory references. Its focus is primarily instruction execution and cache.

4. DERIVED METRICS

Armed with this information we can see that individual event counts, by themselves, are not much help. If a program executes 2 billion floating-point instructions, is that good or bad? What if there were 400 million L2 cache misses? From that data alone it is hard to tell. It is only when counters are combined that a useful picture emerges.

For example, suppose a program running on a single core executes 2 billion double precision floating-point instructions in about 100 million nanoseconds. From this we can compute an execution rate of about 20 gigaflops per second (20 GF/s). It's still not enough information by itself, but further knowing that we have a 2.6 GHz processor that can be accelerated to 3.2 GHz, and an AVX unit capable of 8 floating-point operations per clock, we can determine that the peak rate per core is about 25 GF/s. *Now* the fact that we're achieving 20 GF/s on a single core sounds pretty good. We've hit 80% of peak performance, and from experience we know we're not likely to do much better than that.

Measurements of program performance that are computed from event counts are referred to as "derived metrics". PAPI directly supports a few derived metrics, such as program megaflops, but there are many more that are useful. A fairly complete collection of derived metrics for Sandy Bridge can be found in Appendix B. The list isn't exhaustive, but it captures many of the more important ones.

The importance of derived metrics is that they describe how the program makes use of computer resources. It is true that by themselves they are useful for comparing system performance, but to optimize a program we need to know how we are using system resources relative to its maximum capacity. It's not enough to know that we are computing at a rate of 20 GF/s on a single core, we also have to know that the core is capable of no more than 25 GF/s. Efficiency numbers like that tell us whether there is opportunity for improvement.

There are several metrics that are good indicators of improvement opportunities. Floating-point rates, when shown relative to peak rates, are an obvious choice that is easy to understand. Another good indicator is cache usage. Data in L1 cache can be moved into registers in under two nanoseconds. Moving data from memory to registers requires anywhere from 10 nanoseconds for stride-one access, to 95 nanoseconds for random access. Clearly, the closer the data can be kept to the core, the faster the program will run.

But when it comes to cache, of course, it is never quite that simple. An occasional few references to memory during a very long running program is not going to affect performance all that much, but a large number of memory references can severely impact performance. How do you know when the number of memory loads is limiting performance? Ideally, it is by comparing the amount of time the program is delayed loading data from memory against the total running time of the program. Unfortunately, we don't have counters to measure that.

It *seems* like the next best thing would be to compare the number of load instructions against the execution time. Unfortunately, that doesn't tell us whether the program is being delayed waiting for data from memory or the program is very good at interleaving data operations with other operations. Suppose, for example, that we are doing the worst thing possible, that is, loading lots

of data from random locations in memory. The number of load instructions per unit time would be very low, because each load instruction takes so long to complete. On the other hand, consider a program that is highly efficient and only occasionally loads data from L1 cache. The number of load instructions per unit time would still be low, not because the program is performing poorly, but because it is performing well!

The key issue is whether data load operations are delaying program execution. One way to see this is by comparing the number of memory hits to the total number of instructions over the same period. If there are many memory hits and few instructions executed over a period of time, that's a good indicator that performance is limited by memory performance and perhaps the data layout might benefit from restructuring.

On the other hand, the program might be one that has no other choice but to move lots of data between memory and core. In that case, comparing the memory hits per second against the system maximum memory throughput could be very useful. Each memory hit moves 64 bytes from memory to cache. Furthermore, each processor has one memory controller, each memory controller has four memory channels, each channel is 8 bytes wide, and operates at 1,600 MHz. Thus, each server has a peak memory bandwidth of 2 processors x 4 channels x 8 bytes wide x 1,600 MHz equals 102,400 MB/s, or about 100 GB/s. But there are always inefficiencies in the system, and the best throughput a program can actually achieve is around 78 GB/s. If the program in question *must* move large data blocks around and it does so at or near 78 GB/s, it is doing well.

Another good general indicator of how well memory is managed is the average number of clocks per instruction, or CPI. One core may issue and retire up to four instructions per clock. If the core sustains this peak rate, the CPI will be 0.25. Values close to this can only occur when most data is in registers, branches are accurately predicted, and few conflicts arise between core functional units. When the opposite is true, that is, when data is in memory and memory accesses are random, CPI can reach 150 or more. CPI may also be high when problems unrelated to data motion occur, but CPI will be high when excess data motion occurs.

Two other indicators that are sometimes useful are vector operations per floating-point operation and floating-point operations per load instruction. The first, vector operations per floating-point operation, gives the vectorization rate. Vector operations can be significantly faster than scalar operations. For starters, scalar operations require the processor to issue and retire more instructions than vector operations to accomplish the same work. Vector operations are also better equipped to pipeline the operations, resulting in faster execution. The exception to this rule is when data must come from memory and memory throughput is saturated. In that case, the pipeline must stall waiting for data whether the operations are scalar or vector, and there is no advantage to vector operations.

The second indicator, floating-point operations per load instruction, is sometimes called the arithmetic intensity, or floating-point intensity. It compares the number of desirable operations -- floating-point operations -- against the number of undesirable operations -- memory operations. It is also an indication of the level of data reuse. When re-use is high and data movement is low, performance will also tend to be high. The standard definition of this metric is floating-point

operations per byte of data loaded from memory, but in our case it must be floating-point operations per data load instruction. The two metrics are closely related but different. The event counters do not track the number of bytes loaded, only the number of load instructions, so this alternate definition must be used. The drawback is that load instructions may load from 1 to 32 bytes, so additional knowledge of program behavior may also be needed.

5. INSTRUMENTATION SELECTION

With so many possible events to choose from, selecting the most useful set of events can be challenging. This is made even more challenging by the limited number of counters available.

The way this works is that each processor core has eight event counters available for use. Hyperthreading makes two contexts available per core, so when it is enabled, each of the two Hyperthreads gets half of the counters, or four counters each. Compute servers generally do not benefit from Hyperthreads, though, so they are disabled.

Of the eight counters that are available, the operating system often uses one of the counters for a watchdog timer. A watchdog timer is used to prevent the system from hanging. At certain key times the operating system resets the timer, always long before the timer should expire. If the timer ever does expire, it is because a device driver has hung attempting to access a device that has failed, or a privileged component has failed to manage locks correctly. If a watchdog timer expires, it raises a high priority interrupt to get control back to the operating system, where it can either recover or fail gracefully.

The seven remaining counters are generally available and can be used for nearly any selection of PAPI events. When a counter is initialized, the event type is set, the counter scope is set, and the event count is initialized to a known value. Each time an event of the type specified occurs, the counter is incremented. The counter continues to increment until it rolls over, that is, until the sign of the value changes, at which time it raises an interrupt. The interrupt causes a handler to record the fact that the value rolled over, reset the counter value, and handle any bookkeeping details that might be needed.

Since the number of counters is limited, event selection must be tailored to fit the investigation. For example, if nothing is known about a code, the FLOP rate might be a useful place to start. If only double precision adds and multiplies are counted, it requires only one counter, for `PAPI_DP_OPS` events. Division instructions can be included with an additional counter for `PAPI_FDV_INS`. A third counter is needed to include single precision operations. Elapsed time can be measured using the Linux function `gettimeofday` or the PAPI function `PAPI_get_real_ns`.

The number of memory loads per instruction is also a good place to start. Memory loads generally occur when a load instruction misses all levels of cache. This is recorded by the `PAPI_L3_TCM` event. Each time L3 is missed, a cache line is read from memory into L1 cache, which may cause a cache line to be evicted from L1, L2 and L3. Memory can also be read directly, bypassing cache, using non-temporal read instructions. Non-temporal read operations are *not* counted by `PAPI_L3_TCM`. Total number of instructions is counted by `PAPI_TOT_INS`.

CPI is another good place to start. CPI requires the total number of instructions, using `PAPI_TOT_INS`, and the number of clocks, using `PAPI_TOT_CYC`.

Notice that there is a difference between the total number of cycles, PAPI_TOT_CYC, and the number of reference cycles, PAPI_REF_CYC. Reference cycles are driven by the processor base frequency and occur at a constant rate. They may be measured using an event counter, using the PAPI_REF_CYC event, or by using the Time Stamp Counter, with the PAPI_get_real_cyc function. Either way the result is the same. In contrast, the total number of cycles can vary as the clock frequency goes up and down with the system load. Total cycles are only measured with an event counter and the event PAPI_TOT_CYC.

Vectorization rates can be obtained by collecting the total floating-point operations with PAPI_DP_OPS and the vector operations with PAPI_VEC_DP. Note that PAPI_VEC_DP records the number of floating-point operations that were initiated by a vector instruction, *not* the number of vector instructions issued.

Finally, the number of memory references per instruction can be obtained by gathering PAPI_L3_TCM and PAPI_TOT_INS events.

When only double precision floating-point operations are used, these five metrics can be collected together. The events would be: PAPI_DP_OPS, PAPI_FDV_INS, PAPI_L3_TCM, PAPI_TOT_INS, PAPI_VEC_DP, and PAPI_TOT_CYC. Elapsed time could be measured using the functions PAPI_get_real_cyc and PAPI_get_real_ns.

Note that the above list uses six of the seven available counters, leaving room for one more. L3 hits are expensive, though not so expensive as memory hits, so counting L2 misses using PAPI_L2_TCM might be useful. L3 hits can then be derived from the events collected. Detailed cache studies generally require that several events be counted together, though. Other events, such as branch mispredictions or stalled cycles might also provide some insight. Potentially most useful, though, is PAPI_LD_INS, to provide insight into the number of floating-point operations per load instruction.

6. INSTRUMENTATION PLACEMENT AND DATA GATHERING

Once the PAPI counters have been selected, they must somehow be added to the application, in the locations where they will usefully measure interesting behavior. This means either adding the instrumentation by hand, or using a performance tool, such as TAU or Open Speed Shop, to add it for you. Adding instrumentation by hand can be tedious and error prone, but it gives very precise control over where the instrumentation is placed and when it is activated. Using performance tools can be simpler, but it may also be more difficult to gather the data wanted without extraneous noise. Examples of each will be shown.

6.1. TAU Instrumentation

First we start with an example from TAU.

```
$ bin="../../../bin"
$ module purge
$ . /projects/tau/tau.bashrc
$ module load mkl/15.0
$ /bin/rm -rf MULTI__* profile.*
$ export TAU_METRICS=TIME,PAPI_DP_OPS,PAPI_VEC_DP,PAPI_FDV_INS,PAPI_TOT_INS,\
PAPI_TOT_CYC,PAPI_L3_TCM
$ tau_exec -T serial,papi $bin/daxpy_64_mkl -threads 1 -size 1000 -iterations 10000000
$ paraprof
```

It is important to remove any files named `profile.*` and directories named `MULTI__*`, as that is where the data will be stored. TAU does not take steps to ensure its data collection will take place without difficulty, and pre-existing files with those names can interfere in subtle ways with data gathering and display.

Once the program has run and the data has been gathered, the PAPI counts are displayed using `paraprof`. The initial screen shows the metrics as sub-entries in the navigation pane. An example is shown in Figure 2.

Double-clicking on one of the counters, such as `PAPI_DP_OPS`, opens a new window that shows the range of values for that metric. An example is shown in Figure 3. Unfortunately the data is in graphical form. Double-click on "node 0" to see the numerical value for that metric (Figure 4).

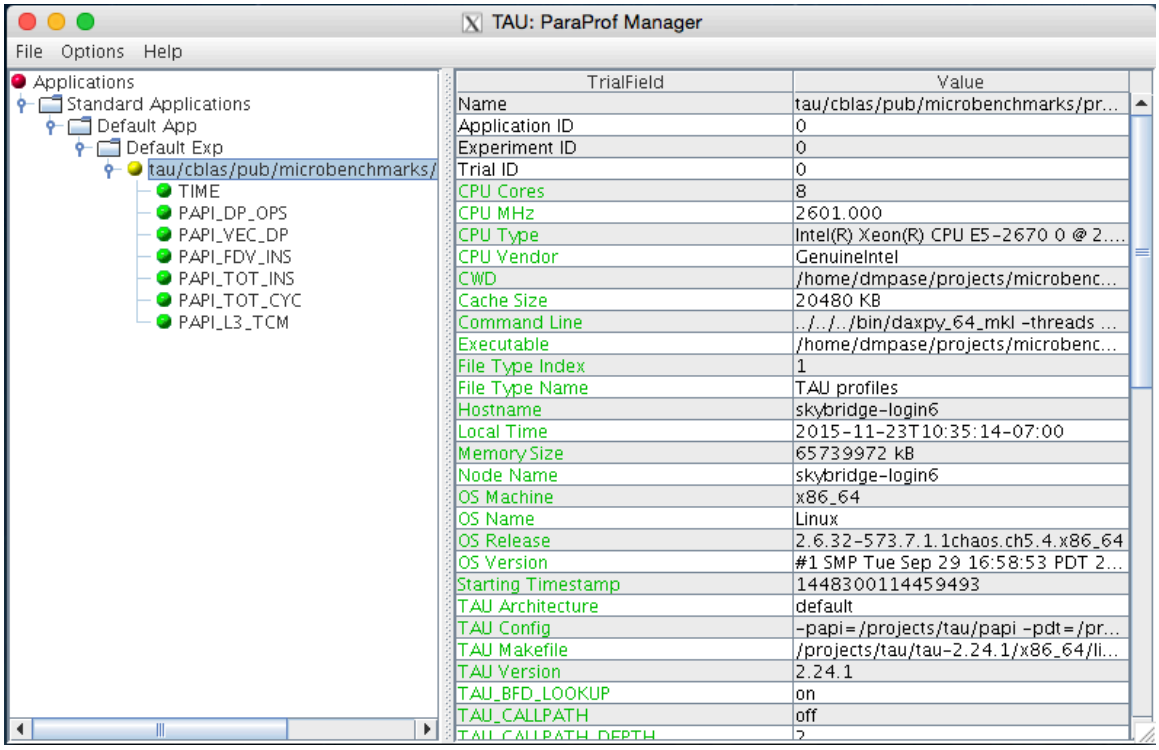

Figure 2 – TAU ParaProf Manager

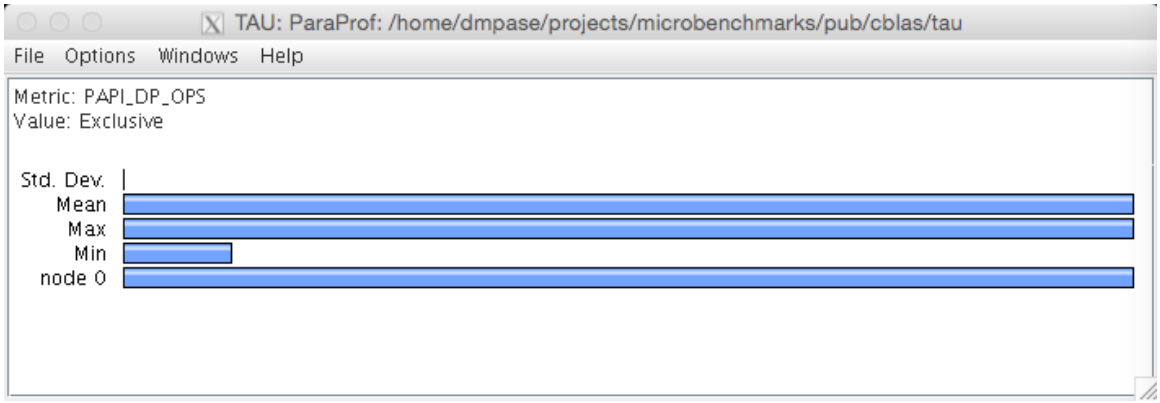

Figure 3 – TAU Metric Window

Figure 4 – TAU Metric Window

There are several down sides to this process. First, collecting data for many counters becomes an exercise in creating and deleting ParaProf windows. You must point and click twice to find the data, copy the data to paper or spreadsheet, then delete the windows you've created. Second, TAU often truncates the number of digits, losing precision that might be useful. Third, while convenient, this form of instrumentation gathers data for the whole program, which may include code you don't want to include in your measurements. TAU has other ways to add PAPI counters into code that offer a little more precision, and they are discussed in detail in TAU documentation.

6.2. Open|SpeedShop Instrumentation

Open|SpeedShop also offers the ability to instrument a code without recompiling it. (See below.)

```
$ module add tools/openss-openmpi
$ osshwcsamp "$bin/daxpy_64_mkl -threads 1 -size 1000 -iterations 10000000" \
PAPI_DP_OPS,PAPI_TOT_CYC,PAPI_FDV_INS,PAPI_L3_TCM,PAPI_TOT_INS,PAPI_VEC_DP

[openss]: hwcsamp experiment using input papi event:
"PAPI_DP_OPS,PAPI_TOT_CYC,PAPI_FDV_INS,PAPI_L3_TCM,PAPI_TOT_INS,PAPI_VEC_DP".
[openss]: hwcsamp experiment using the hwc experiment default sampling_rate: "100".
[openss]: hwcsamp experiment calling openss.
[openss]: Setting up offline raw data directory in ./offline-oss
[openss]: Running offline hwcsamp experiment using the command:
"../../../bin/daxpy_64_mkl -threads 1 -size 1000 -iterations 10000000"

Routine     Elements Bytes/Thread Time (sec) Iterations          MFLOPS            MB/s
-------|------------|------------|----------|----------|---------------|---------------
  daxpy         1000        16000     2.8789   10000000       6947.1154      83365.3850


[openss]: Converting raw data from ./offline-oss into temp file X.0.openss

Processing raw data for daxpy_64_mkl ...
Processing processes and threads ...
Processing performance data ...
Processing symbols ...
Resolving symbols for /home/dmpase/projects/microbenchmarks/bin/daxpy_64_mkl
Updating database with symbols ...
Finished ...

[openss]: Restoring and displaying default view for:
      /home/dmpase/projects/microbenchmarks/pub/cblas/oss/daxpy_64_mkl-hwcsamp-10.openss
[openss]: The restored experiment identifier is:  -x 1

Exclusive    % of CPU  papi_dp_ops  papi_tot_cyc  papi_fdv_ins  papi_l3_tcm  papi_tot_ins
papi_vec_dp  Function (defining location)
```

```
 CPU time        Time
       in
 seconds.
 2.860000   99.651568  24497864411   9386176728          138         145  24360778635
24497859092  cblas_daxpy (daxpy_64_mkl: cblas_daxpy.c,12)
 0.010000    0.348432     85633584     32824677                           85163493
85633584  main (daxpy_64_mkl: daxpy.c,40)
 2.870000  100.000000  24583497995   9419001405          138         145  24445942128
24583492676  Report Summary
```

The Open|SpeedShop output is somewhat difficult to read when copied to this page, but it's a little easier on a wide terminal. Open|SpeedShop does limit the number of PAPI counters to six, even when seven counters are available in the system.

6.3. Manual Instrumentation

Adding instrumentation by hand to an application is rarely a preferred method for instrumenting a code. It can be tedious and error prone, not only to add the instrumentation when first gathering data, but also when removing it after it is no longer needed. It is more suitable as a method of last resort. However, at least in simple cases, PAPI instrumentation is easy to add and remove. All that is required is an array of events to count and a few simple routines. For example, in the DAXPY program example, the code looks something like this:

```
#include <papi.h>
int events[] = {
    PAPI_DP_OPS,               // vector and scalar DP ops
    PAPI_FDV_INS,              // floating-point divisions
    PAPI_VEC_DP,               // vector DP ops
    PAPI_L3_TCM,               // L3 total cache misses
    PAPI_TOT_INS,              // total instructions
    PAPI_TOT_CYC,              // total cycles (TurboBoosted)
};
const int len = sizeof events / sizeof *events;

long long *vals = (long long *) malloc(len * sizeof(long long));
PAPI_start_counters(events, len);

for (i=0; i < iterations; i++) {
    cblas_daxpy(size, scalar, x, 1, y, 1);
}

PAPI_stop_counters(vals, len);
```

This code works for a single thread, but more work would be required for this to handle multithreaded code correctly. A particularly valuable advantage, though, is that the output can be designed to suit any preferences. Not a big deal when the number of interesting counters is small, but PAPI supports 50 counters on Sandy Bridge and for that, or when the number of instrumented regions is large, the convenience can be significant.

7. PERFORMANCE ANALYSIS

Once the data has been collected, it's time to figure out what's really going on in the application. To help illustrate what one might expect to see with these counters, we ran four benchmarks with a selection of carefully chosen input parameters. The benchmarks are: DAXPY, to show the effects of strided memory access; PCHASE, to show the effects of random memory access; and DGEMM and DGETRF, to show high performance floating-point codes. Detailed descriptions of the benchmarks and their performance characteristics are given in Appendix C.

7.1. DAXPY

In a nutshell, DAXPY loads two vectors, multiplies the first one by a scalar, adds the vectors together and stores the sum back into the second vector. For each vector element, it loads two double precision floating-point values and stores one. It has very few floating-point operations for each reference to memory. References are all stride-one, that is, they hit successive locations in memory. That makes it very easy for a cache prefetch engine to predict where the next reference will hit, which allows the engine to load the next cache line while the current line is being used.

The vector size can be used to predict where in the cache hierarchy the data will be found. Each double precision floating-point vector element is 8 bytes, so two vectors of length 1,000 uses 16,000 bytes. This fits nicely within the 32 KiB L1 data cache and two vectors of length 10,000 fit within L2 cache. Vectors of length 100,000 and 1,000,000 fit within L3 cache, while larger vectors fit only within memory.

DAXPY has yet one more advantage: it can be easily executed as either vector code or scalar code, allowing us to see how the counters differentiate between the two.

For this exercise, we used vector lengths of 1,000 up to 1,000,000,000 elements. For each vector length, we gathered counter values for the cache counters `PAPI_L1_TCM`, `PAPI_L2_TCM`, `PAPI_L3_TCM`, and for the instruction counters `PAPI_DP_OPS`, `PAPI_FDV_INS`, `PAPI_TOT_CYC`, `PAPI_TOT_INS`, and `PAPI_VEC_DP`. Counter values for the scalar and vector versions were gathered separately. The values are recorded in Appendix D, Tables D.1 and D.3. The derived metrics discussed in previous sections are also recorded in Appendix D, Tables D.2 and D.4.

Now that the data has been gathered, the next step in this analysis is to verify that the `PAPI_DP_OPS` event collects useful data. Starting with floating-point operations, Tables D.1 and D.3 show the number of DP OPS to be approximately 20,000,000,000. We expect the number of DP OPS to be 2 x length x iterations, which in every case is also 20,000,000,000, so the counter appears to be accurate. Furthermore, Figure 5 shows the gigaflops per second calculated from PAPI counters, which matches the blue (single core) performance curve in Figure C.1.

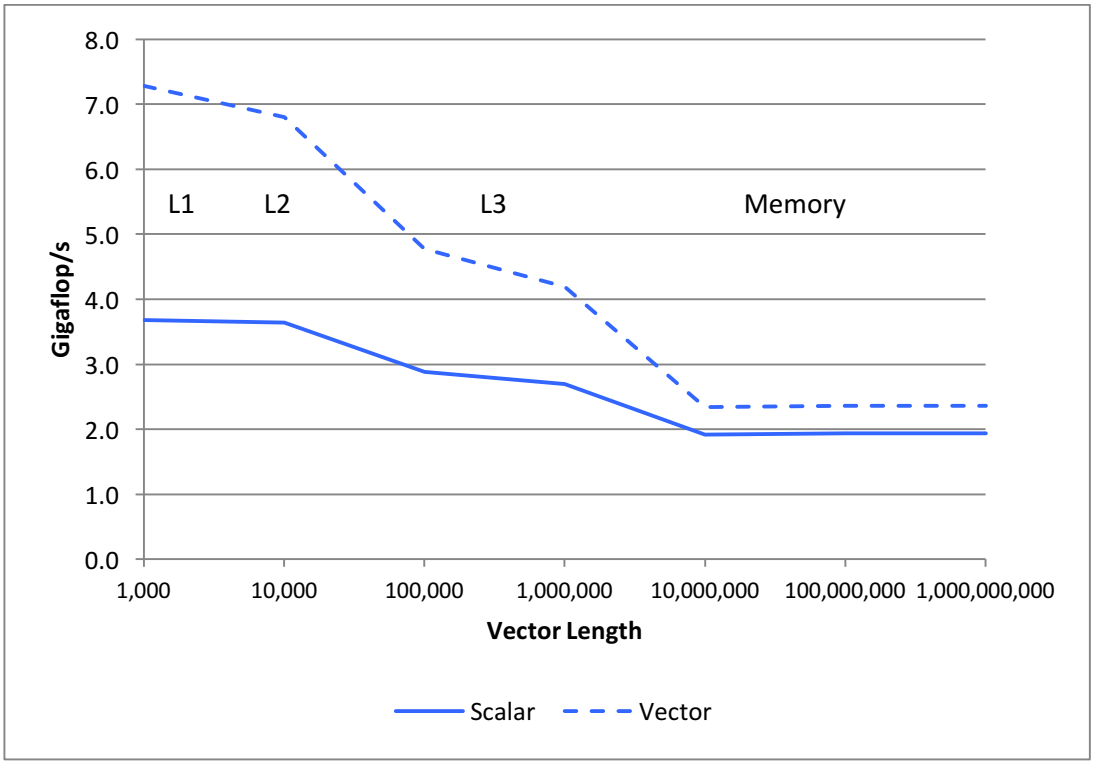

Figure 5 – DAXPY Gigaflops Per Second

From Figure 5 we see that the number of gigaflops per second is well below what the processor AVX floating-point unit is capable of, which we calculated earlier to be about 25 GF/s/core. Whatever else might be going on, we know the AVX unit isn't the limiting factor here. We aren't floating-point performance limited.

If the benchmark *had* been FP performance limited, we would look to systems with faster clocks, faster vector units, or more cores (greater parallelism).

What's more, we can see that performance declines as the vectors become larger. While we know *a priori* that this is due to data fitting into various levels of cache, we will be able to see it directly from later measurements.

Our next step is to look at the rate that instructions are issued and retired. We will actually use its inverse - Clocks Per Instruction. This data is shown in Figure 6. CPI is based on the `PAPI_TOT_INS` event. The measurement of 45,000,000,000 total instructions for 20,000,000,000 scalar floating-point instructions seems plausible, although we have no way to verify this more closely.

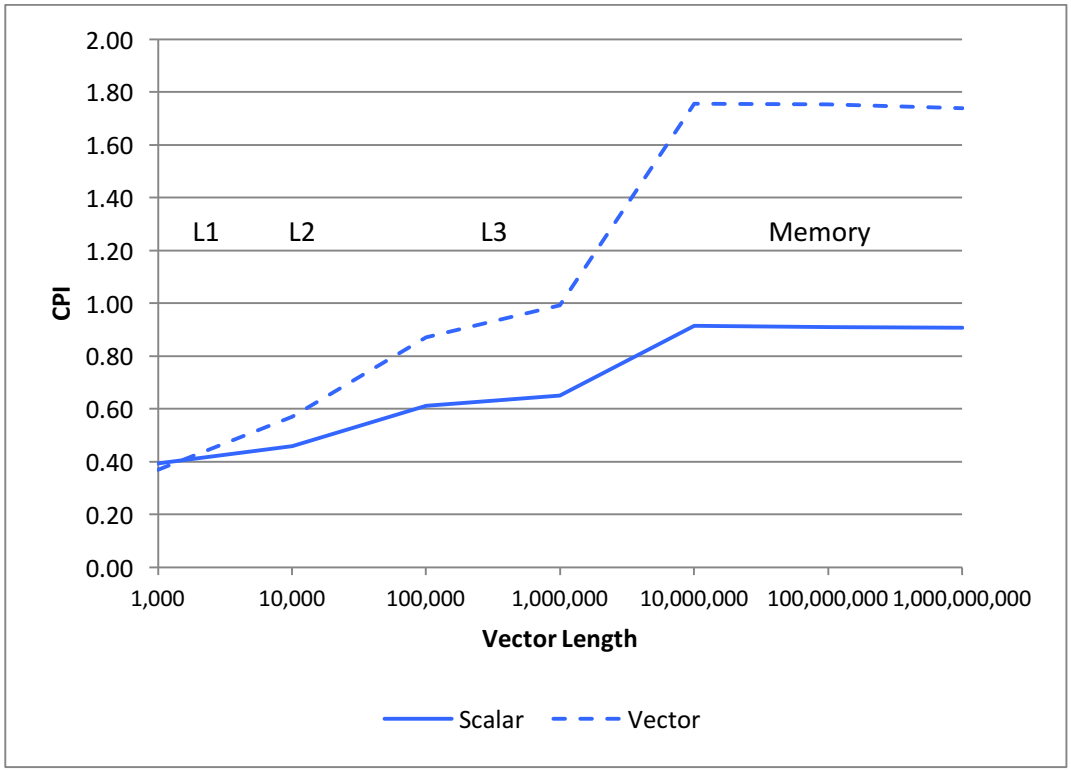

Figure 6 – DAXPY CPI

CPI tells us whether the core itself is running at full speed. We've determined that the floating-point units aren't fully utilized. Is that because the integer units, or some other components, are heavily used? If so, this could affect us in any of several ways. One way is the units that issue and retire instructions may be fully loaded. There's work to do, the data is there, but the core just isn't able to go any faster. Another is that there is simply a lot of integer work to do, and it's getting the bulk of the core.

At most the core can issue/retire 4 instructions per clock. If the CPI is close to its theoretical limit of 0.25, we know that the core is operating at full speed. Essentially, we would be core limited, also referred to as being clock frequency limited. If that were the case, then to go faster we would need faster clocks or more cores.

We can see from Figure 6 that we are not core limited. We also see that the core becomes progressively idle as the vectors become longer. Our next step will be to examine the contribution of the memory subsystem, to see how it affects performance.

Ideally we would be able to measure how much time the core is delayed waiting for data from cache, but nothing like that is available. Instead we have cache miss counts for each level of cache. Those events are `PAPI_L1_TCM`, `PAPI_L2_TCM` and `PAPI_L3_TCM`. There are other related events that could also be used, but these three event types give a count of all cache misses, including instructions and data, for each level of cache.

It's important to note that each cache miss represents a cache line to be loaded into cache. New cache lines are always loaded into L1, which may cause a ripple effect of data being evicted from each layer of cache.

In order to see whether our performance is impacted by data being away from the core, we look at the number of instructions executed, on average, between cache misses. This information is shown in Figure 7.

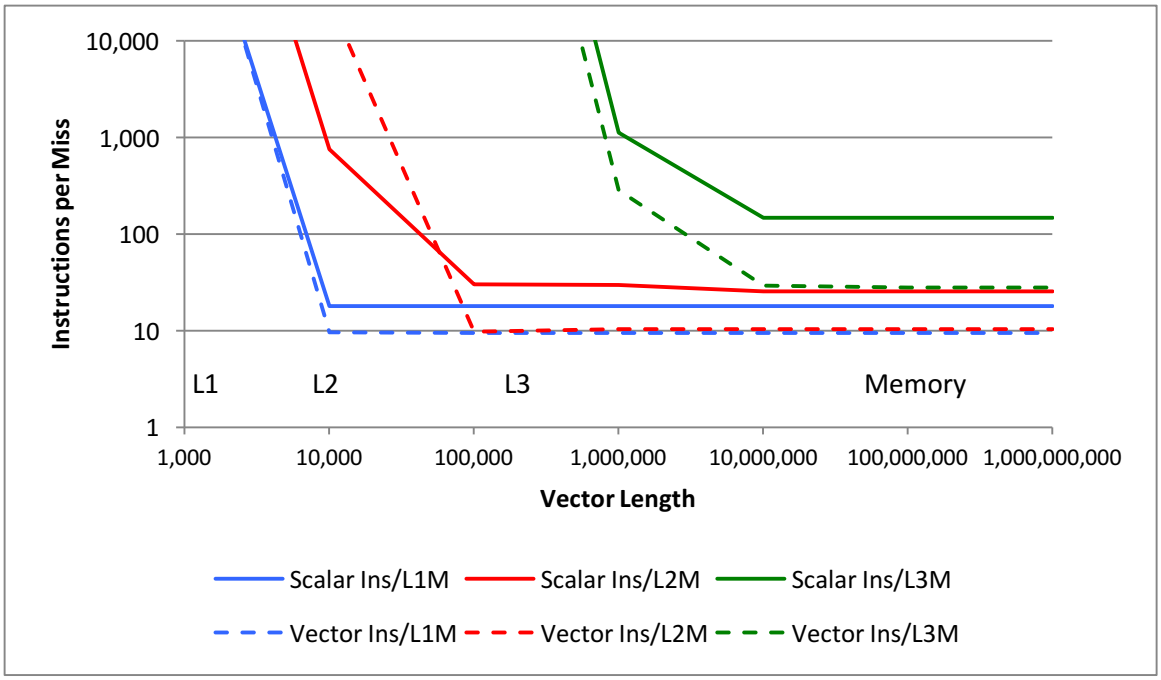

Figure 7 – DAXPY Instructions Retired Per Cache Miss

We can see several things from looking at Figure 7. When the vector length is 1,000, the number of instructions per miss is very high for all three levels of cache. This is a strong indication that cache misses are not impacting performance. This is as we expect, because we know that the data fits completely in L1 cache.

When the vector length is 10,000, the number of instructions per miss is low for L1 cache (blue lines), but high for L2 and L3 caches (red and green lines). This means we are missing L1 often, but hitting in L2 and L3. We know this is true because we know the vectors fit in L2 cache. But had this been an unfamiliar piece of code, we could also see it from the data. This same pattern is also repeated for L3.

Restructuring a code to increase cache reuse is a complex topic, but these three metrics clearly show when there may be an opportunity to improve program performance by increasing cache hits.

There are several other generally useful metrics. Two of them are the vectorization rate and the "Turbo GHz". The vectorization rate measures how well a code is vectorized. Not surprisingly,

our two examples of DAXPY show opposite extremes. The scalar version has a vectorization rate of zero, while our vectorized version shows a rate of 100%. Vectorization rate is computed using PAPI_DP_OPS, which counts all double precision adds and multiplies, and PAPI_VEC_DP, which counts only vector DP operations. (Divisions are not counted either way.)

"Turbo GHz" (our term) is a measure of how much the processor clock is allowed to increase while staying within the core thermal limits (aka Thermal Design Point, or TDP). It can be shown as a ratio of total clocks to reference clocks, or as an effective clock frequency. The effective clock frequency tells you what frequency, on average, the core is *really* operating at.

7.2. PCHASE

The PCHASE benchmark measures memory random access latency by repeatedly chasing a chain of pointers distributed randomly throughout a block of memory. The benchmark is described in greater detail in Appendix C, but the most important point to understand is that each link in the chain loads a new cache line into cache. Each pointer, each link in the chain, is placed so as to be completely unpredictable to any cache prefetch engine, forcing the full latency for each pointer fetch. No other operations are performed, other than to check for the end of the chain.

Furthermore, chains are formed from a contiguous block of memory, and the size of the memory block determines its cache residency. These tests used sizes ranging from 12,800 bytes to 12,800,000,000 bytes, with each test having 10x more links in the pointer chain and 10x fewer iterations over the chain, than the previous test. The first test fits entirely in L1 cache, the next test in L2, followed by two tests in L3, and the remaining tests in local memory.

Our first step with this benchmark is to make a quick consistency check on the counters. We gather the same 8 events as before and look for reasonable counts. By increasing the chain length and decreasing the iteration count simultaneously, the total number of instructions should be very close from one run to the next. The inner loop itself consists of a load instruction, an increment, and a conditional branch. Furthermore, the chain length times the iteration count always equals 2,000,000,000, so there should be about 6,000,000,000 instructions counted. Looking at Table D.5 we see that PAPI_TOT_INS count is approximately that.

The floating-point counters are easy to check. PAPI_DP_OPS, PAPI_FDV_INS and PAPI_VEC_DP are all near zero, as they should be. They aren't all exactly zero, but relative to the number of instructions executed, they are "close enough".

The cache events, PAPI_L1_TCM, PAPI_L2_TCM and PAPI_L3_TCM, are the most interesting events in the list. The critical value is the chain length times the iteration count, as that gives the number of cache lines to be loaded for each benchmark run. As mentioned, that value is 2,000,000,000 in all cases. The total number of cache misses for all three caches for the test run that fits in L1 cache (size is 12,800) is very close to zero. The test that fits into L2 cache shows roughly 2,000,000,000 L1 cache misses, and nearly zero misses for L2 and L3. The two L3 tests show 2,000,000,000 L1 and L2 cache misses, and low miss counts for L3. The memory

tests show approximately 2,000,000,000 cache misses for all three levels of cache. It appears that each of these event types is working correctly in this case.

Next we examine the counters as if we were analyzing the performance of an unfamiliar program, and compare that with what we know about the program. We start with CPI. From Figure 8 (and Table D.6), we see the CPI in all cases is well above the value of 0.25 we would like to see. From this we know that we are not limited by core performance and would suspect we are limited by the time to fetch data from cache or memory (which in this case we know is true).

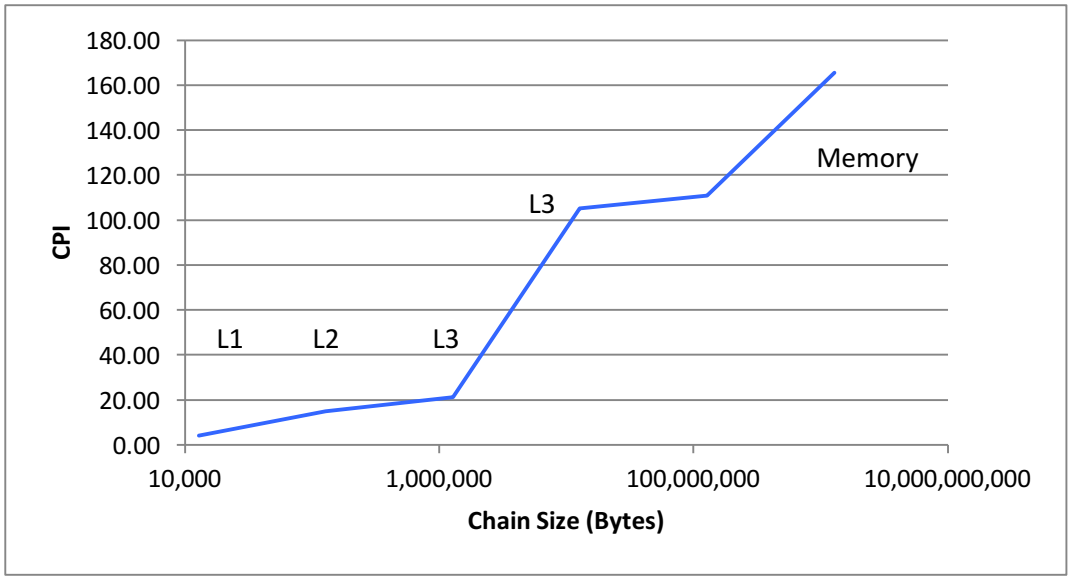

Figure 8 – PCHASE CPI

Since CPI is very high, we look at the cache miss behavior. In particular, we look at the number of instructions per cache miss for each of the levels of cache.

When the chain size is small enough to fit in the L1 cache (12,800 bytes), Figure 9 shows that the average number of instructions between L1 (blue line), L2 (red line), and L3 (green line) misses is very large. In essence, for the 12,800-byte test, missing cache at any level did not affect performance. That being the case, we could also look at the `PAPI_LD_INS` event and compare it against the `PAPI_L1_TCM` counter. With 2,000,000,000 load instructions and very few L1 cache misses, the loads are hitting L1 and that is what limits performance, especially when we consider 1 instruction in 3 is a load instruction.

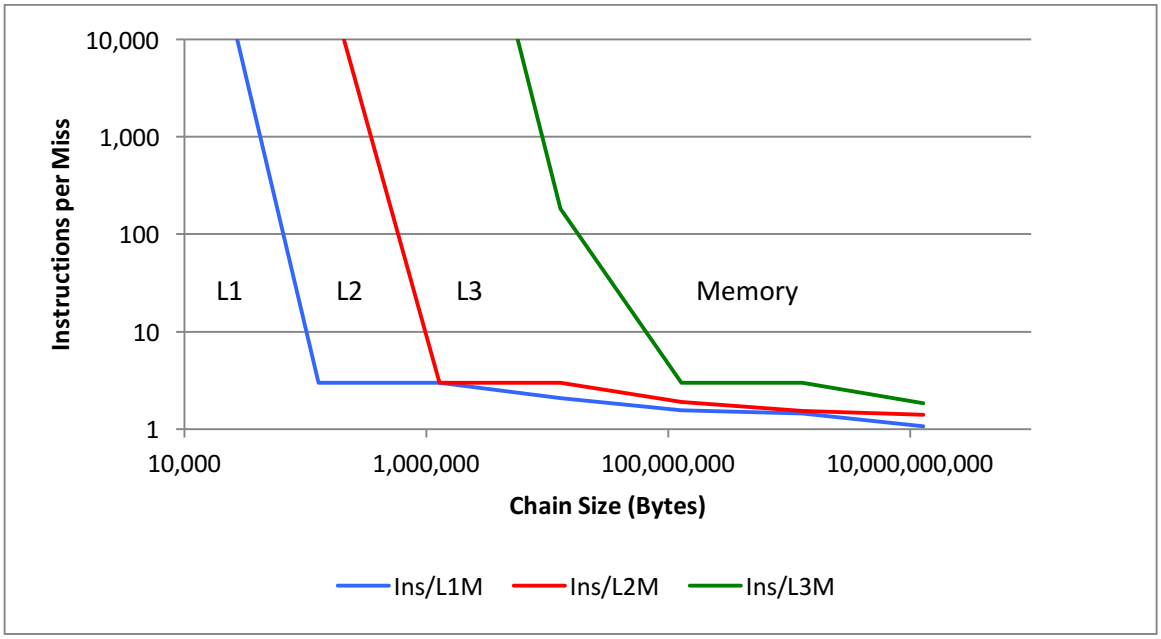

Figure 9 – PCHASE Instructions Retired Per Cache Miss

When the chain size is 128,000 bytes and all data fits in L2, we see a similar effect. However, since the chain no longer fits into L1, the number of instructions executed per L1 miss drops to 3, while the number of instructions executed per L2 or L3 miss remains very high. This tells us immediately that we are hitting in L2. Again, since 1 instruction in 3 is a load instruction, it is the speed of L2 that limits performance. Similarly, when data is in L3 cache, the number of instructions per L2 miss drops to 3 while the number of instructions per L3 miss remains high. When data is too large for the L3 cache, the number of instructions per cache miss drops to very low numbers for all three levels of cache.

Optimizing a program whose performance is limited by the performance of random memory access requires significant restructuring of the data in memory. Data must be moved into lower levels of cache and reused extensively. How this might be done depends on the nature of the program and is beyond the scope of this paper.

7.3. DGEMM

So far we have examined a program that strides through memory, DAXPY, and a program that randomly accesses memory, PCHASE. The next two programs, DGEMM and DGETRF, are highly optimized and make very efficient use of the core floating-point unit. We know the number of floating-point operations they perform and that the number of load operations is proportional to the number of floating-point operations. We will use the PAPI event counters to see what we can learn about these two benchmarks.

DGEMM is a routine from the Level 3 BLAS, or Basic Linear Algebra Subroutine package. It multiplies two matrixes together and adds the result to a third matrix. Details of this operation are described in Appendix C. This routine is able to achieve very high performance over a wide variety of matrix sizes, exceeding 95% of the theoretical maximum double precision floating-

point performance in many cases. The actual floating-point performance is shown in Figure 10. We see that the performance rapidly approaches 25 gigaflops/second. Since we know the single core double precision floating-point performance peak is just over 25 gigaflops/second, we can see that we are limited by core performance.

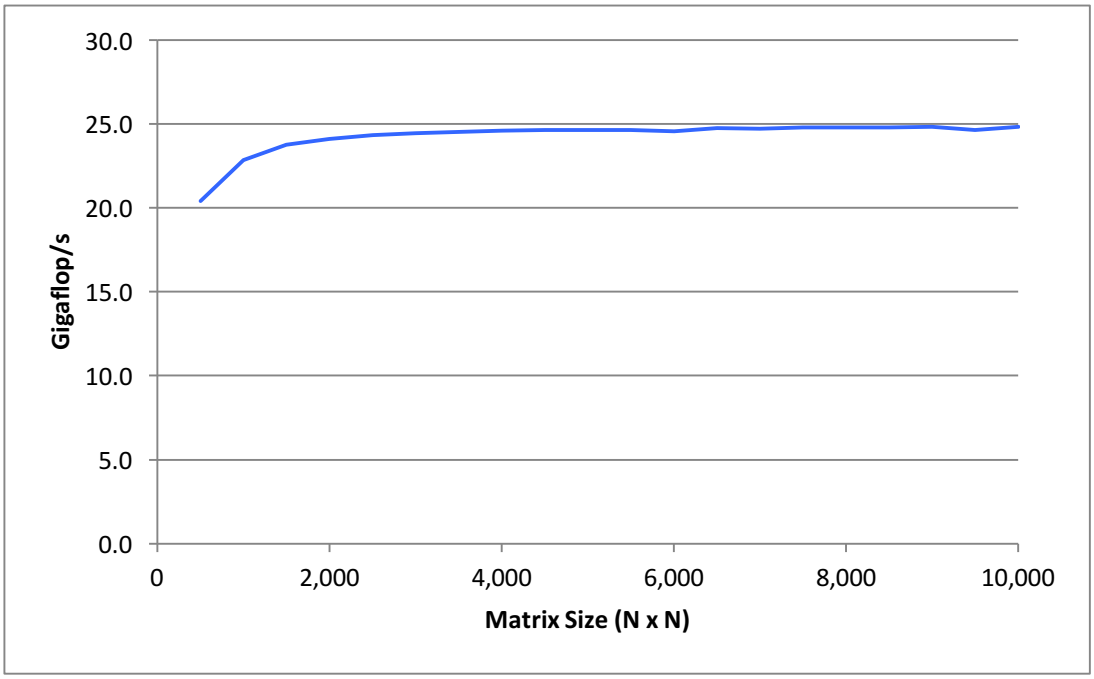

Figure 10 – DGEMM Gigaflops Per Second

CPI is also very flat and low, at approximately 0.35. (See Figure 11.) This means that three out of every four instruction slots have an instruction available for execution. This is a very high rate.

The vectorization rate across the board is at 100%.

Once again we look at the rate of instructions retired per cache miss to see whether improvements might be possible. But from Figure 12 we see that cache access is quite balanced and infrequent enough that cache misses are not a limiting factor. The number of instructions executed before an L1 miss occurs is about 40 instructions. An L1 miss that is satisfied in L2 requires about 12 clock cycles, which gives plenty of time to load data into registers. The number of instructions before an L2 miss is about 140, compared to an L2 miss satisfied in L3, which is about 45 clock cycles.

It is interesting to note that 40:12 and 140:45 are both about 3:1, which is the number of instructions per clock we see from the CPI (i.e., 1/0.35 = 2.85). This may be an indication that the cache is a slight limiting factor to performance.

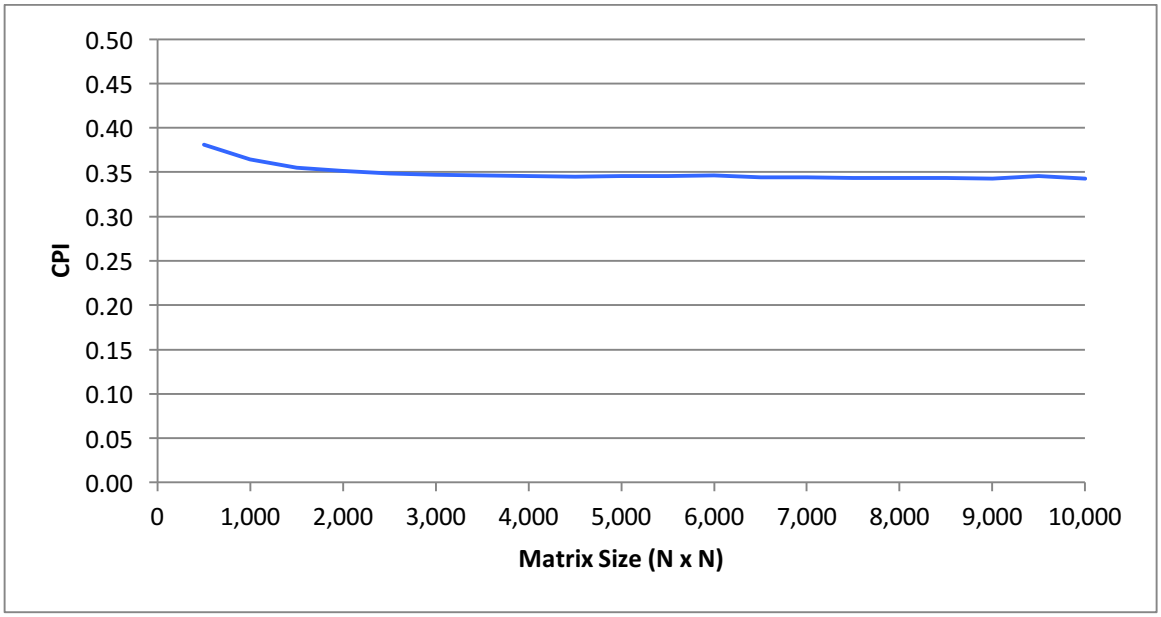

Figure 11 – DGEMM CPI

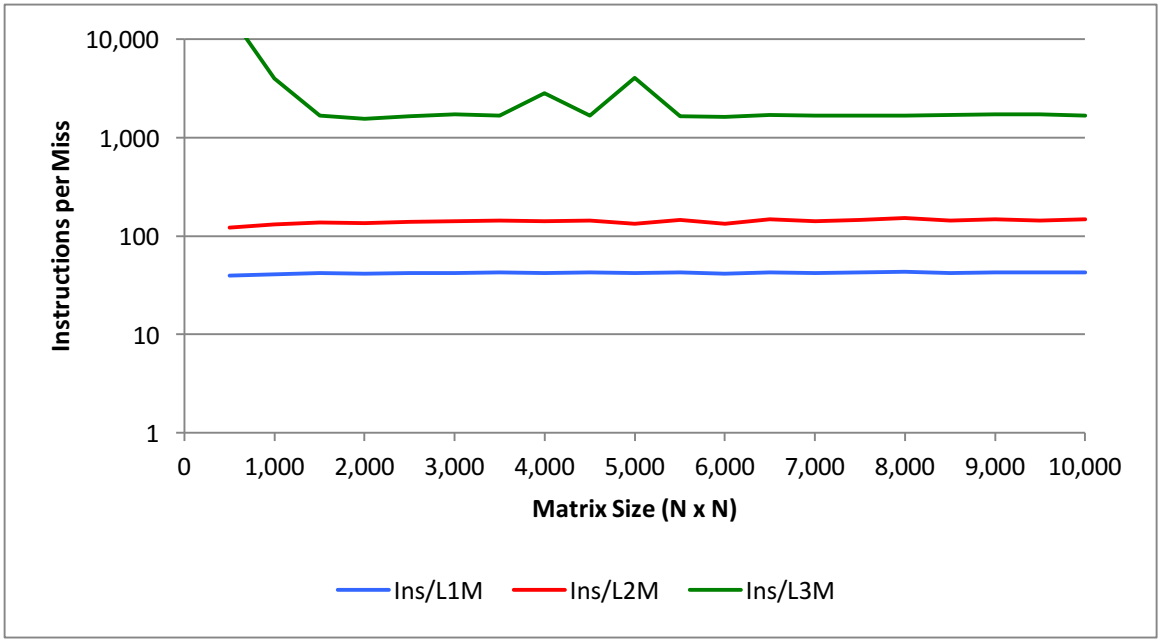

Figure 12 – DGEMM Instructions Retired Per Cache Miss

There are about 1,660 instructions before an L3 miss, and an L3 miss of strided memory costs around 10 nanoseconds. Clearly 1,660 instructions take more than enough time to prefetch data into L3. The performance of memory does not appear to be a limiter to performance based on this evidence.

Given this information, it appears the best ways to increase performance here are a faster vector floating-point unit, or possibly a larger L2 or L3 cache. This code is obviously making very efficient use of the available hardware.

7.4. DGETRF

DGETRF is a subroutine from the linear algebra package library called LAPACK. It solves a system of *N* linear equations in *N* unknowns using a technique known as LU factorization with partial pivoting. The system of equations is represented as a dense, rectangular matrix. Like DGEMM, DGETRF also makes efficient use of an available core, although not quite to the same degree. We know the number of floating-point operations that DGETRF performs for any given size of system. We also know that the number of load operations is proportional to the number of floating-point operations, and that ratio of floating-point to load operations is very small.

As with the other benchmarks, raw event counts and derived metrics are given in Appendix D. DGETRF event counts are in Table D.9 and derived metrics are in Table D.10.

We start our analysis with the floating-point performance (Figure 13) and see that floating-point performance increases as *N* increases. In this case we reach about 22.5 gigaflops per second when *N* is 14,000, which represents about 88% of peak performance.

CPI (Figure 14) shows a mirror image of the floating-point performance because floating-point operations dominate this benchmark. As *N* becomes large enough, CPI approaches 0.35, which is the same CPI number we saw for DGEMM. But while the CPI performance might be the same, the floating-point performance is not, because DGETRF requires more integer operations to manage internal details of the algorithm.

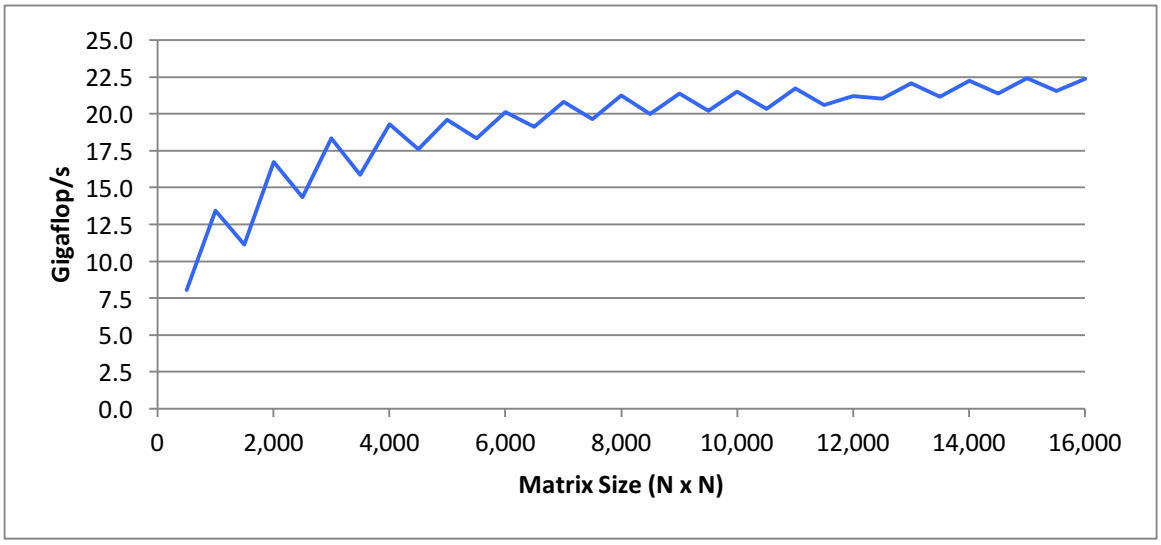

Figure 13 – DGETRF Gigaflops Per Second

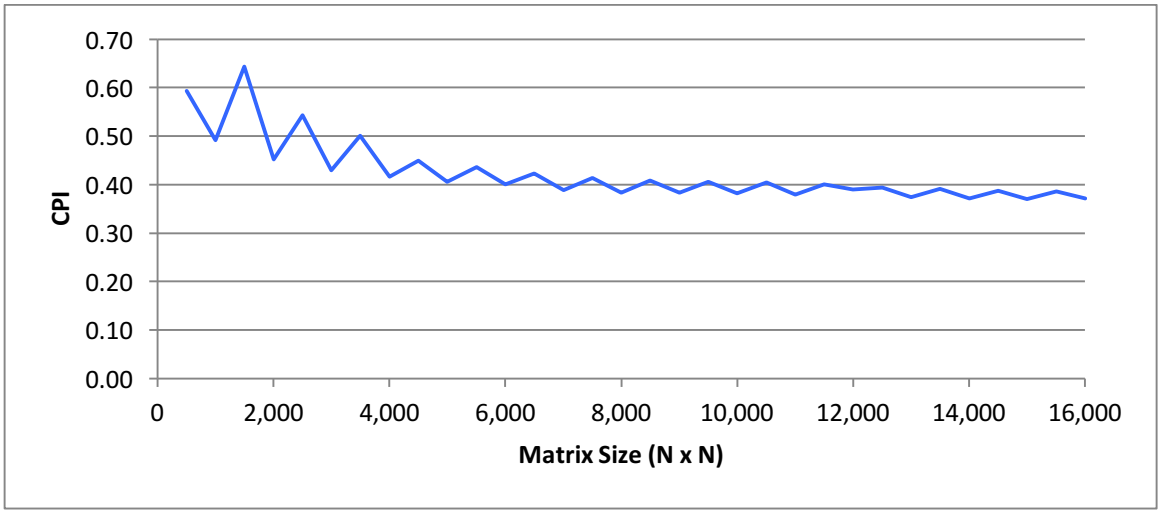

Figure 14 – DGETRF CPI

The cache behavior (Figure 15 or Table D.10) is also very similar to DGEMM, though perhaps not quite as favorable. Looking at the number of instructions before an L1 cache miss, DGETRF executes about 30 to 40 instructions before an L1 miss, 80 to 150 instructions before an L2 miss, and 900 to 1,400 instructions before an L3 miss. If instructions and data references were perfectly overlapped, there is just enough time that neither one becomes a major bottleneck.

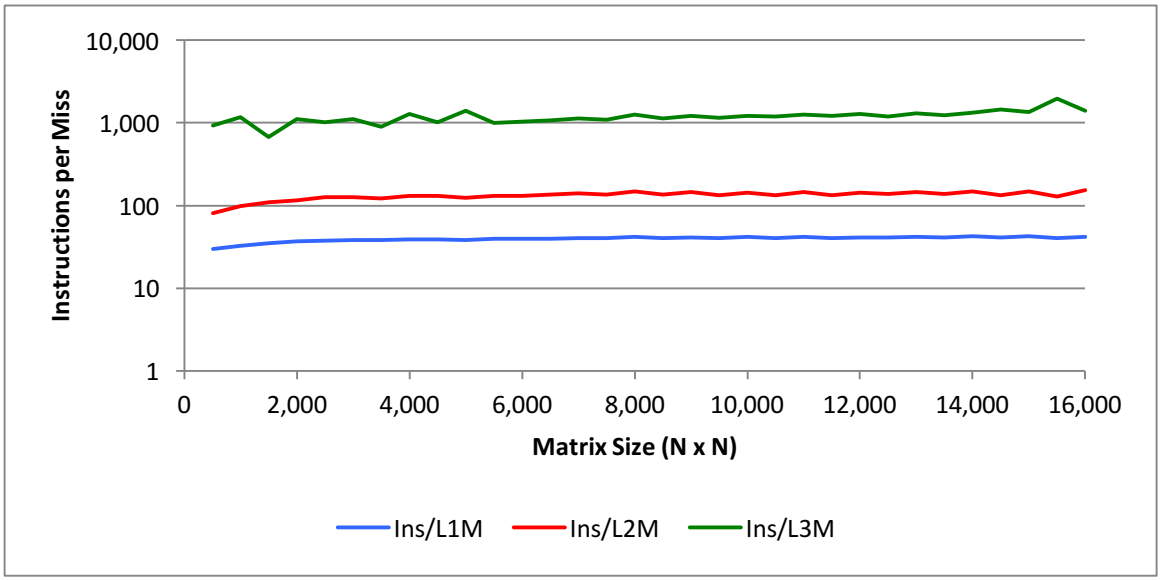

Figure 15 – DGETRF Instructions Retired Per Cache Miss

Like DGEMM, it appears that DGETRF balances its demands on the major core components to achieve high performance. The floating-point performance is high, the CPI is low, and data references hit cache and memory with a low enough frequency that they do not impede performance.

8. SUMMARY

PAPI is a useful means for obtaining low-level performance data. The Intel Sandy Bridge processor implements eight event counters per core, of which up to seven are available to system users. Many different event types can be counted, but knowing what those counters represent, or how to use them to analyze the performance of an application, can be difficult. Furthermore, Intel has acknowledged, at least in private communication, that some event types are not implemented correctly.

In this document we have verified that a selection of the most useful counters do give consistent and reasonable results, at least for our limited tests. We have also demonstrated ways of using those counters for analyzing the performance of an application, from selecting the counters, to inserting them into an application, to interpreting the results.

APPENDIX A - PAPI EVENTS

Name: PAPI_BR_CN
Units: Instructions
Description: Conditional branch instructions
Notes: Counts the total number of conditional branch instructions retired, including those from `if`, `for`, `while`, and `switch` constructs.

Name: PAPI_BR_INS
Units: Instructions
Description: Branch instructions
Notes: Counts the total number of branch instructions retired, including conditional branches, unconditional branches, subroutine calls and subroutine exits.

Name: PAPI_BR_MSP
Units: Instructions
Description: Conditional branch instructions mispredicted
Notes: Counts the total number of conditional branch instructions that were mispredicted. When a processor core encounters a conditional branch instruction, it guesses which direction the branch will take and speculatively executes along that branch. If the guess is correct, the core has saved time by overlapping instructions along the branch with the time required to evaluate the condition. If the guess is wrong, though, the core must undo anything it has done along the wrong branch and begin to execute along the correct branch. This event counts the guesses that are wrong.

Name: PAPI_BR_NTK
Units: Instructions
Description: Conditional branch instructions not taken
Notes: Counts the number of conditional branches not taken. When a processor core encounters a conditional branch, it has a choice of either taking the branch or falling through to the next instruction. This event counts the number of conditional branches that fall through to the next instruction.

Name: PAPI_BR_PRC
Units: Instructions
Description: Conditional branch instructions correctly predicted
Notes: Counts the total number of conditional branch instructions that were mispredicted. When a processor core encounters a conditional branch instruction, it guesses which direction the branch will take and speculatively executes along that branch. If the guess is correct, the core has saved time by overlapping instructions along the branch with the time required to evaluate the condition. If the guess is wrong, though, the core must undo anything it has done along the wrong branch and begin to execute along the correct branch. This event counts the guesses that are right.

Name: PAPI_BR_TKN
Units: Instructions
Description: Conditional branch instructions taken
Notes: Counts the number of conditional branches taken. When a processor core encounters a conditional branch, it has a choice of either taking the branch or falling through to the next instruction. This event counts the number of conditional branches that do not fall through to the next instruction.

Name: PAPI_BR_UCN
Units: Instructions
Description: Unconditional branch instructions
Notes: Counts the number of unconditional branches executed. Unconditional branches include subroutine calls and exits, and the branches at the bottom of loops and that separate `else` clauses in conditional constructs.

Name: PAPI_DP_OPS
Units: Double precision floating-point operations
Description: Floating point operations; optimized to count scaled double precision vector operations
Notes: Counts all double precision addition and multiplication floating-point operations, whether from scalar or vector instructions. It does not include division operations.

Name: PAPI_FDV_INS
Units: Floating-point division instructions
Description: Floating point divide operations
Notes: Counts all floating-point division operations, whether single or double precision.

Name: PAPI_FP_INS
Units: Floating-point instructions
Description: Floating point instructions
Notes: Counts all single precision and double precision scalar operations, but not vector operations. Note that for scalar instructions, instructions and operations are the same thing. Note also that this event gives the same results as PAPI_FP_OPS.

Name: PAPI_FP_OPS
Units: Floating-point operations
Description: Floating point operations
Notes: Counts all single precision and double precision scalar operations, but not vector operations. Note that for scalar instructions, instructions and operations are the same thing. Note also that this event gives the same results as PAPI_FP_INS.

Name: PAPI_L1_DCM
Units: 64-byte cache lines
Description: Level 1 data cache misses
Notes: Counts the number of accesses that miss, causing a 64-byte cache line to be read-in to the L1 data cache. Accesses include both load and store operations. Thus a program with stride-one access and data not in the L1 cache will have one data cache miss out of eight 8-byte load operations it performs. Note that this event gives very similar results as PAPI_L2_DCA.

Name: PAPI_L1_ICM
Units: 64-byte cache lines
Description: Level 1 instruction cache misses
Notes: Counts the number of instructions that miss, causing a 64-byte cache line to be read-in to the L1 instruction cache.

Name: PAPI_L1_LDM
Units: 64-byte cache lines
Description: Level 1 load misses
Notes: Counts the number of load operations that miss the L1 cache. Each load miss causes a 64-byte cache line to be read-in to the L1 data cache.

Name: PAPI_L1_STM
Units: 64-byte cache lines
Description: Level 1 store misses
Notes: Counts the number of store operations that miss the L1 cache. A cache store miss causes a 64-byte cache line to be read-in to the cache, then modified by the store operation, before the operation completes.

Name: PAPI_L1_TCM
Units: 64-byte cache lines
Description: Level 1 cache misses
Notes: Counts all misses, including all loads and stores, to the L1 data and instruction caches. Note that this event should give results that are very similar to PAPI_L2_TCA.

Name: PAPI_L2_DCA
Units: 64-byte cache lines
Description: Level 2 data cache accesses
Notes: Counts all accesses, including all loads and stores, to the L2 cache. (Intel uses a unified L2 data/instruction cache.) This includes all accesses whether they hit or miss the cache. Note that this event gives results that are very similar to PAPI_L1_DCM.

Name: PAPI_L2_DCH
Units: 64-byte cache lines
Description: Level 2 data cache hits
Notes: Counts the number of hits to the L2 cache. (Intel uses a unified L2 data/instruction cache.) Each hit results from an L1 miss and causes a 64-byte cache line to be copied from L2 to the L1 data cache. Any subsequent references to that data will be satisfied from L1 until it is flushed from the L1 cache. Data copied to L1 may cause a cache line to be evicted from L1, which may then be stored in L2, causing an eviction to L3, which may, in turn, cause an L3 eviction to memory.

Name: PAPI_L2_DCM
Units: 64-byte cache lines
Description: Level 2 data cache misses
Notes: Counts the number of misses to the L2 cache. (Intel uses a unified L2 data/instruction cache.) Each miss causes a 64-byte cache line from L3 cache or from memory. Each L2 miss ultimately results from an L1 miss, so the cache line is copied into L1 cache. Intel uses a non-exclusive cache design, so the cache line may or may not reside in more than one level of cache at any time. Note that this event gives very similar results as PAPI_L3_DCA.

Name: PAPI_L2_DCR
Units: 64-byte cache lines
Description: Level 2 data cache reads
Notes: Counts the number of reads to the L2 cache. (Intel uses a unified L2 data/instruction cache.) L2 cache reads result from L1 cache misses, so each read results in a 64-byte cache line being written to the L1 cache. If the cache line resides in L2 cache, it is copied from L2 to L1. If it does not, it is brought in from L3 or from memory.

Name: PAPI_L2_DCW
Units: 64-byte cache lines
Description: Level 2 data cache writes
Notes: Counts the number of L2 cache writes, whether or not the data actually resides in L2 cache. In a write-back cache configuration, which is most common, temporal write operations cause a cache line read to L1, if needed, followed by an update to the cache line using the written data. Afterwards, the modified cache line resides in L1. However, introducing a new cache line to L1 may evict a cache line to L2. This is the source of L2 cache writes. An L2 cache write may also cause a cache line to be evicted to L3, and subsequently an L3 cache line may be evicted to memory. Note that Intel uses a unified L2 data/instruction cache design, so this event will give very similar results to PAPI_L2_TCW.

Name: PAPI_L2_ICA
Units: 64-byte cache lines
Description: Level 2 instruction cache accesses
Notes: Counts the number of L2 accesses that result from instruction fetches. An instruction access results from an L1 instruction cache miss, and causes a 64-byte cache line to be copied to the L1 instruction cache.

Name: PAPI_L2_ICH
Units: 64-byte cache lines
Description: Level 2 instruction cache hits
Notes: Counts the number of L2 hits that result from instruction fetches. An instruction hit results from an L1 instruction cache miss, and causes a 64-byte cache line to be copied from L2 to the L1 instruction cache.

Name: PAPI_L2_ICM
Units: 64-byte cache lines
Description: Level 2 instruction cache misses
Notes: Counts the number of L2 misses that result from instruction fetches. An instruction miss results from an L1 instruction cache miss, and causes a 64-byte cache line to be copied from L3 or from memory.

Name: PAPI_L2_ICR
Units: 64-byte cache lines
Description: Level 2 instruction cache reads
Notes: Counts the number of L2 reads that result from instruction fetches. An instruction read results from an instruction fetch that misses the L1 instruction cache. If the cache line resides in L2, it is copied to the L1 instruction cache. If it does not, it is copied to L1 from L3 or memory.

Name: PAPI_L2_STM
Units: 64-byte cache lines
Description: Level 2 store misses
Notes: Counts the number of store operations that miss the L2 cache. L2 store operations that miss have also missed the L1 cache, and the cache line must be copied to L1 from L3 or memory.

Name: PAPI_L2_TCA
Units: 64-byte cache lines
Description: Level 2 total cache accesses
Notes: Counts the number of total accesses to L2 cache. Accesses include both load and store operations, whether or not the data resides in cache at the time. L2 accesses generally result from L1 cache misses, and often result in data being copied to L1 cache. However, the use of non-temporal loads and stores can cause data to be accessed in L2 without being copied to L1. Note that this event should give very similar results as PAPI_L1_DCM.

Name: PAPI_L2_TCM
Units: 64-byte cache lines
Description: Level 2 total cache misses
Notes: Counts the number of misses, both loads and stores, to L2 cache. L2 misses are the result of L1 misses, and generally result in data being copied from L3 or memory to L1 cache. An L1 miss can be from either L1 data cache or L1 instruction cache. Note that this event should give very similar results as PAPI_L3_TCA.

Name: PAPI_L2_TCR
Units: 64-byte cache lines
Description: Level 2 total cache reads
Notes: Counts the total number of L2 cache reads, whether or not the data actually resides in L2 cache. L2 reads are the result of an L1 read miss, and result in a 64-byte cache line being sent to L1 cache.

Name: PAPI_L2_TCW
Units: 64-byte cache lines
Description: Level 2 total cache writes
Notes: Counts the total number of L2 cache writes, whether or not the data actually resides in L2 cache. In a write-back cache configuration, which is most common, temporal write operations cause a cache line read to L1 followed by an update to the cache line using the written data. Afterwards, the modified cache line resides in L1. However, introducing a new cache line to L1 may evict a cache line to L2. An L2 cache write may also cause a cache line to be evicted to L3, and subsequently an L3 cache line may be evicted to memory. Note that Intel uses a unified L2 data/instruction cache design, so this event will give very similar results to PAPI_L2_DCW.

Name: PAPI_L3_DCA
Units: 64-byte cache lines
Description: Level 3 data cache accesses
Notes: Counts all accesses, including all loads and stores, to the L3 cache. (Intel uses a unified L3 data/instruction cache.) This includes all access whether they hit or miss the cache. Note that this event often gives very similar results to PAPI_L2_TCM, because an L3 access rarely occurs without an L2 miss.

Name: PAPI_L3_DCR
Units: 64-byte cache lines
Description: Level 3 data cache reads
Notes: Counts the number of reads to the L3 cache. (Intel uses a unified L3 data/instruction cache.) L3 cache reads result from L2 cache misses, so each read results in a 64-byte cache line being written to the L2 cache. If the cache line resides in L3 cache, it is copied from L3 to L2. If it does not, it is brought in from memory.

Name: PAPI_L3_DCW
Units: 64-byte cache lines
Description: Level 3 data cache writes
Notes: Counts the total number of L3 cache writes, whether or not the data actually resides in L3 cache. In a write-back cache configuration, which is most common, temporal write operations cause a cache line read to L1 followed by an update to the cache line using the written data. Afterwards, the modified cache line resides in L1. However, introducing a new cache line to L1 may evict a cache line to L2, which may also cause a cache line to be evicted to L3, and subsequently an L3 cache line may be evicted to memory. Note that Intel uses a unified L3 data/instruction cache design, so this event will give very similar results to PAPI_L3_TCW.

Name: PAPI_L3_ICA
Units: 64-byte cache lines
Description: Level 3 instruction cache accesses
Notes: Counts the number of L3 accesses that result from instruction fetches. An instruction access results from an L2 instruction read miss, and causes a 64-byte cache line to be copied to the L1 instruction cache.

Name: PAPI_L3_ICR
Units: 64-byte cache lines
Description: Level 3 instruction cache reads
Notes: Counts the number of L3 reads that result from instruction fetches. An instruction read results from an instruction fetch that misses the L2 instruction cache. If the cache line resides in L3, it is copied to the L1 instruction cache. If it does not, it is copied from memory.

Name: PAPI_L3_TCA
Units: 64-byte cache lines
Description: Level 3 total cache accesses
Notes: Counts the number of total accesses to L3 cache. Accesses include both load and store operations, whether or not the data resides in cache at the time. L3 accesses generally result from L2 cache misses, and often result in data being copied to L1 cache. However, the use of non-temporal loads and stores can cause data to be accessed in L3 without being copied to L1. Note that this event should give very similar results as PAPI_L2_TCM.

Name: PAPI_L3_TCM
Units: 64-byte cache lines
Description: Level 3 total cache misses
Notes: Counts the number of misses, both loads and stores, to L3 cache. L3 misses are the result of L1 and L2 misses, and generally result in data being copied from memory to L1 cache. An L1 miss can be from either L1 data cache or L1 instruction cache. Note that this event indicates that memory was read, although it gives no indication whether it was local or remote memory.

Name: PAPI_L3_TCR
Units: 64-byte cache lines
Description: Level 3 total cache reads
Notes: Counts the total number of L3 cache reads, whether or not the data actually resides in L3 cache. L3 reads are the result of an L1 and L2 read miss, and result in a 64-byte cache line being sent to L1 cache.

Name: PAPI_L3_TCW
Units: 64-byte cache lines
Description: Level 3 total cache writes
Notes: Counts the total number of L3 cache writes, whether or not the data actually resides in L3 cache. In a write-back cache configuration, which is most common, temporal write operations cause a cache line read to L1 followed by an update to the cache line using the written data. Afterwards, the modified cache line resides in L1. However, introducing a new cache line to L1 may evict a cache line to L2, which may also cause a cache line to be evicted to L3, and subsequently an L3 cache line may be evicted to memory. Note that Intel uses a unified L3 data/instruction cache design, so this event will give very similar results to PAPI_L3_DCW.

Name: PAPI_LD_INS
Units: Instructions
Description: Load instructions
Notes: Counts the number of load instructions executed. Note that this is the number of instructions and not the number of operands fetched, so a vector load represents a single event even though it may fetch up to four operands.

Name: PAPI_REF_CYC
Units: Processor core clock cycles
Description: Reference clock cycles
Notes: Counts the number of processor core base clock cycles, also known as the core reference cycles. This reflects the nominal frequency of the core before any TurboBoost acceleration takes place. It remains constant even when clock acceleration takes place. It does not require an actual event counter, but comes from the Time Stamp Counter. This value is obtained by calling PAPI_get_real_cyc().

Name: PAPI_REF_NS
Units: Nanoseconds
Description: Reference nanoseconds
Notes: Counts the number of nanoseconds that transpires. Like the PAPI_REF_CYC counter, it does not require an event counter but comes from the Time Stamp Counter instead. This value is obtained by calling PAPI_get_real_nsec().

Name: PAPI_SP_OPS
Units: Single precision floating-point operations
Description: Floating point operations; optimized to count scaled single precision vector operations
Notes: Counts all single precision addition and multiplication floating-point operations, whether from scalar or vector instructions. It does not include division operations.

Name: PAPI_SR_INS
Units: Instructions
Description: Store instructions
Notes: Counts the number of store instructions executed. Note that this is the number of instructions and not the number of operands stored, so a vector store represents a single event even though it may store up to four operands.

Name: PAPI_STL_ICY
Units: Processor core clock cycles
Description: Cycles with no instruction issue
Notes: Counts the number of processor core clock cycles that were stalled, that is, no new instructions could be issued because of either resource conflicts within the ALUs or the next instruction block was pending arrival from memory or cache.

Name: PAPI_TLB_DM
Units: Instructions
Description: Data translation lookaside buffer misses
Notes: Counts the number of instructions that caused Data Translation Look-Aside Buffer (DTLB) misses. The TLB caches physical address ranges and speeds up the process of converting a virtual address to a physical address. TLB misses increase the latency of fetching or storing data to memory.

Name: PAPI_TLB_IM
Units: Instructions
Description: Instruction translation lookaside buffer misses
Notes: Counts the number of instructions that caused Translation Look-Aside Buffer (TLB) misses when fetching the instructions from memory. The TLB caches physical address ranges and speeds up the process of converting a virtual address to a physical address. TLB misses increase the latency of fetching or storing data to memory.

Name: PAPI_TOT_CYC
Units: Processor core clock cycles
Description: Total cycles
Notes: Counts the total number of processor core clock cycles completed. This count includes the additional cycles that occur when the clock frequency is increased through TurboBoost.

Name: PAPI_TOT_INS
Units: Instructions
Description: Instructions completed
Notes: Counts the total number of instructions retired. This includes all types of instructions, such as floating-point and integer instructions, branch instructions, etc.

Name: PAPI_VEC_DP
Units: Double precision vector floating-point operations
Description: Double precision vector/SIMD instructions
Notes: Counts the total number of double precision vector floating-point operations (**not instructions**) executed. In purely vector codes, PAPI_VEC_DP and PAPI_DP_OPS give similar values.

Name: PAPI_VEC_SP
Units: Single precision vector floating-point operations
Description: Single precision vector/SIMD instructions
Notes: Counts the total number of single precision vector floating-point operations (**not instructions**) executed. In purely vector codes, PAPI_VEC_SP and PAPI_SP_OPS give similar values.

APPENDIX B - DERIVED METRICS

Name: Processor Core Base Frequency
Formula: `PAPI_REF_CYC / PAPI_REF_NS`
Description: Nominal processor core frequency before TurboBoost, in GHz
Notes: May be used to verify or determine the processor base frequency. Larger values are usually better.

Name: Double-Precision FLOP Rate (GFLOPS or GFLOP/s)
Formula: `(PAPI_DP_OPS + PAPI_FDV_INS) / PAPI_REF_NS`
Description: Double precision floating-point performance, in gigaflops/second
Notes: Computes the floating-point performance, in double precision floating-point operations per second. Larger values are usually better.

Name: Clocks Per Instruction (CPI)
Formula: `PAPI_TOT_CYC / PAPI_TOT_INS`
Description: Core instruction retirement rate, in cycles per instruction
Notes: Computes the number of instructions retired per accelerated processor clock. The Intel Architecture is able to issue and retire up to 4 instructions per processor clock period, so the smallest this value can be is 0.25. In the other extreme, a program that is constantly stalled waiting for memory can see this value in the hundreds. Smaller values are usually better.

Name: Processor Turbo Ratio
Formula: `PAPI_TOT_CYC / PAPI_REF_CYC`
Description: Performance increase due to TurboBoost
Notes: Intel processors support increasing the clock frequency to improve performance, as long as doing so remains within the required thermal envelope. Vector floating-point instructions tend to push the thermal envelope, while memory and other instructions tend to cool the processor. The increased clock frequency is reflected in the total number of elapsed cycles, while the nominal frequency is reflected in the reference cycles. The ratio of these two values gives the percentage of clock improvement. Larger values are usually better.

Name: Processor Turbo Frequency
Formula: `PAPI_TOT_CYC / PAPI_REF_NS`
Description: Accelerated processor core frequency due to TurboBoost, in GHz
Notes: Intel processors support increasing the clock frequency to improve performance, as long as doing so remains within the required thermal envelope. Vector floating-point instructions tend to push the thermal envelope, while memory and other instructions tend to cool the processor. The increased clock frequency is reflected in the total number of elapsed cycles, while the nominal frequency is reflected in the reference cycles. The ratio of total cycles to reference nanoseconds gives the effective processor clock frequency. Larger values are usually better.

Name: Ratio of Load Instructions
Formula: `PAPI_LD_INS / PAPI_TOT_INS`
Description: Ratio of load instructions to all instructions
Notes: Measures the ratio of load instructions to all instructions in the measured range. Instructions that move memory, load and store instructions, are often viewed as a necessary expense to set up a computation rather than useful work. They are also expensive in that they require anywhere from 3 clocks for an L1 cache hit, to many hundreds of clocks to load data from remote memory. This metric gives an indication of how intensely the memory hierarchy is used. Smaller values are usually better, but higher values are not necessarily bad. If the CPI is high and this ratio is also high, then performance may be limited by the many references to memory.

Name: Time Per Load Instruction
Formula: `PAPI_REF_NS / PAPI_LD_INS`
Alternate: `PAPI_TOT_CYC / PAPI_LD_INS`
Description: Time in nanoseconds per load instruction
Notes: Measures the time between load instructions. Instructions that move memory, load and store instructions, are often viewed as a necessary expense to set up a computation rather than useful work. They are also expensive in that they require anywhere from 3 clocks for an L1 cache hit, to many hundreds of clocks to load data from remote memory. This metric gives an indication of how intensely the memory hierarchy is used. Smaller values are usually better, but higher values are not necessarily bad. If the CPI is high and this ratio is also high, then performance may be limited by the many references to memory.

Name: Vectorization Rate
Formula: `PAPI_VEC_DP / (PAPI_DP_OPS + PAPI_FDV_OPS)`
Description: Floating-point vectorization rate
Notes: Measures the fraction of floating-point instructions that are vector instructions. Vector instructions are generally faster than scalar instructions, so a higher percentage of vector instructions is generally desirable. However, it is also worth noting that the effectiveness of vectorization depends on how effectively the data can be kept near the processor. Vector operations when data is in registers and L1 cache show significantly higher performance than similar scalar operations. On the other hand, when data is in memory, vector operations show no advantage over similar scalar operations.

Name: Ratio of Floating-Point Instructions
Formula: `PAPI_DP_OPS / PAPI_TOT_INS`
Description: Ratio of floating-point instructions to all instructions
Notes: Programs generally consist of instructions that do the intended work, instructions that set up the work, and control instructions. For many programs, floating-point instructions do the desired work, while instructions are considered overhead. High FLOP rates can only be achieved when many of the instructions executed are floating-point instructions. So, larger values are usually better.

Name: Floating-Point Intensity or Arithmetic Intensity
Formula: `PAPI_DP_OPS / PAPI_LD_INS`
Description: Floating-point operations per load instruction
Notes: Programs generally consist of instructions that do the intended work and other instructions, overhead, that set up the work to be done. While there are several types of overhead, data movement is potentially the most expensive. Loading data from a random location in memory can take many hundreds of clock cycles, compared to a floating-point operation that takes only a few. For best performance, there should be many operations per load, and data should be close, so larger values are usually better. Something to be aware of, though, is that load instructions may bring in anything from a single byte to a vector of several words, and that is not reflected in the number of load instructions. But since actual data movement is in units of cache lines, this may be less of a problem than it might appear to be.

Name: Memory Hits Per Load
Formula: `PAPI_L3_TCM / PAPI_LD_INS`
Description: Cache lines read from memory per load instruction
Notes: Load operations from memory are expensive. A typical memory load takes 80 to 95 nanoseconds, or about 200 to 250 processor clocks. High memory hits per load, coupled with high CPI values and a high ratio of loads per instruction, indicate the number of memory references may limit performance. Fewer memory hits are usually better.

Name: Memory Hits Per Instruction
Formula: `PAPI_L3_TCM / PAPI_TOT_INS`
Description: Cache lines read from memory per instruction
Notes: Load operations from memory are expensive. A typical random memory load takes 80 to 95 nanoseconds, or about 200 to 250 processor clocks. A typical strided load may take about 10 nanoseconds, because the prefetch engine starts the load before it is actually requested. Either way it takes relatively few memory hits before memory speed dominates program performance. As such, smaller values for this metric are usually better.

Name: L3 Cache Hits
Formula: `PAPI_L2_TCM - PAPI_L3_TCM`
Description: Cache line requests satisfied in L3 cache
Notes: L3 cache hits are generally expensive, about 17.5 ns, or 45 clocks, per hit, though much less expensive than hits to memory. This metric indicates the number of cache lines loaded from L3 cache. Smaller values are usually better.

Name: L3 Cache Hits Per Load
Formula: `(PAPI_L2_TCM - PAPI_L3_TCM) / PAPI_LD_INS`
Description: Cache lines read from L3 cache per load instruction
Notes: Load operations from L3 cache are expensive, although less expensive than memory. A typical random L3 cache load takes 15 to 18 nanoseconds, or about 38 to 45 processor clocks. Strided loads take closer to 7 nanoseconds. High L3 hits per load, coupled with high CPI values or high loads per instruction, indicate the number of L3 cache references may limit performance. Fewer L3 hits are usually better.

Name: L3 Cache Hits Per Instruction
Formula: `(PAPI_L2_TCM - PAPI_L3_TCM) / PAPI_TOT_INS`
Description: Cache lines read from L3 cache per instruction
Notes: Load operations from L3 cache are expensive, although less expensive than memory. A typical random L3 cache load takes 15 to 18 nanoseconds, or about 38 to 45 processor clocks. Strided loads take closer to 7 nanoseconds. High L3 hits per instruction indicate the number of L3 cache references may limit performance. Fewer L3 hits are usually better.

Name: L2 Cache Hits
Formula: `PAPI_L1_TCM - PAPI_L2_TCM`
Description: Cache line requests satisfied in L2 cache
Notes: L2 cache hits are moderately inexpensive, about 4.65 ns, or 12 to 15 clocks, per hit. This metric indicates the number of cache lines loaded from L2 cache.

Name: L2 Cache Hits Per Load
Formula: `(PAPI_L1_TCM - PAPI_L2_TCM) / PAPI_LD_INS`
Description: Cache lines read from L2 cache per load instruction
Notes: Load operations from L2 cache are moderately inexpensive. A typical random L2 cache load takes 4.65 nanoseconds, or about 12 to 15 processor clocks. Strided loads take closer to 3.5 nanoseconds. High L2 hits per load, coupled with high CPI values or high loads per instruction, indicate the number of L2 cache references may limit performance.

Name: L2 Cache Hits Per Instruction
Formula: `(PAPI_L1_TCM - PAPI_L2_TCM) / PAPI_TOT_INS`
Description: Cache lines read from L2 cache per instruction
Notes: Load operations from L2 cache are moderately inexpensive. A typical random L2 cache load takes 4.65 nanoseconds, or about 12 to 15 processor clocks. Strided loads take closer to 3.5 nanoseconds. High L2 hits per instruction indicate the number of L2 cache references may limit performance.

APPENDIX C - BENCHMARK DETAILS

DAXPY

The DAXPY benchmark consists of a simple timed loop that repeatedly computes the vector operation `y[i]=a*x[i]+y[i]`. The values `a`, `x` and `y` are all double precision floating-point values. Access is stride-one. The size of the vectors `x` and `y` determines whether the operations will take place from L1, L2, or L3, or from memory. For vectors of size *N*, the amount of memory used is 2 vectors x 8 bytes per element x *N* elements per vector. The number of operations is 2 x *N* for each complete iteration through the vectors. Thus DAXPY is a reasonable proxy for approximating certain program behavior when dense vectors and matrixes are used.

DAXPY performance is limited almost entirely by the speed of fetching data from the memory hierarchy. Floating-point operations can be issued every clock, but at best data can be fetched from L1 cache, which takes from 3 to 5 clocks per reference. At 3.2 GHz (effective boosted frequency of a 2.6 GHz E5-2670), the best scalar performance is 3.9 GF/s and the best vector performance is 7.4 GF/s per core. It drops off quite severely when data only fits in L2 or L3 cache, or in memory. This performance is clearly shown in Figure C.1.

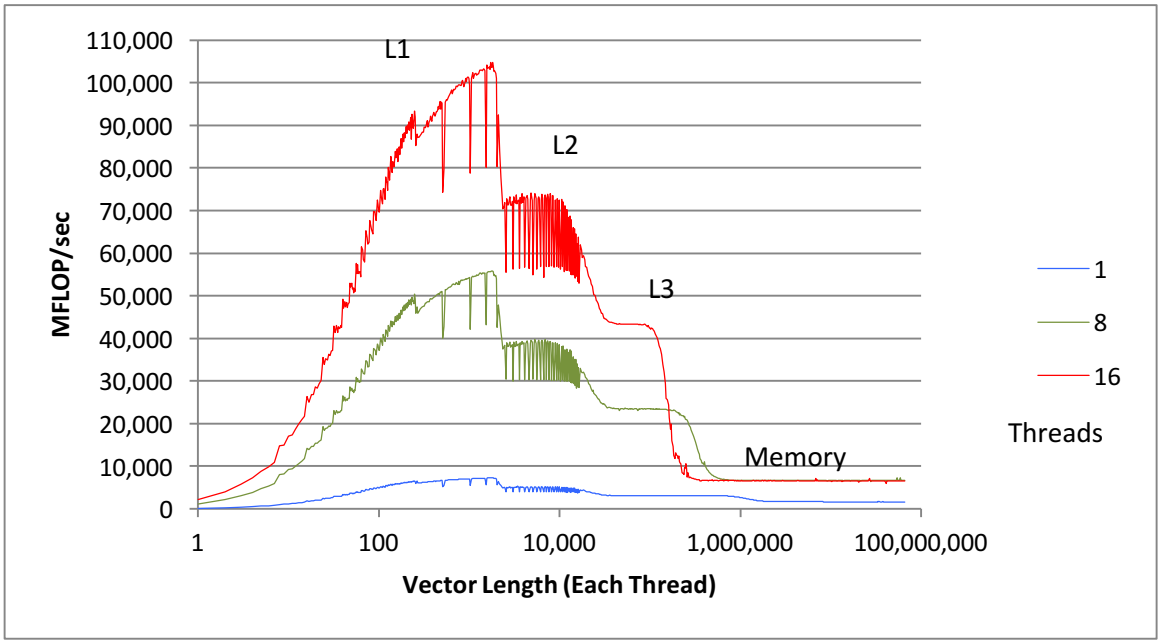

Figure C.1 – DAXPY Performance Using Vector Operations (Intel MKL)

The effect of the memory hierarchy is clearly seen in this diagram. The first performance spike occurs when all data is in L1 cache. The performance slopes upward as the overhead of calling the routine is gradually amortized over longer vector lengths. Performance peaks at around 1440 elements per vector, which is 22.5 KiB of memory for the two vectors, or just under the 32 KiB size of the L1 data cache. Beyond that point both vectors fit in L2 cache, but not L1, and performance falls to the second plateau.

The second plateau occurs where all data fits in the 256 KiB L2 cache. Interestingly, the plateau begins its decline when the two vectors reach about 8,000 elements (128 KiB), or half of the L2 cache. The author has no explanation why the transition begins here, but between this size and about 28,000 elements the data is split between the L2 and L3 caches. Performance for this plateau is about 70% of the peak, when all data fits in L1 cache.

The third plateau begins when all data fits into L3 cache and is no longer found in the L2. The L3 is shared across cores, so fewer cores in use means a larger plateau. The performance at this plateau is about 40% of the peak.

The final plateau begins when data no longer fits into L3 cache and is found only in memory. Note that the performance of this final plateau depends on the number of cores in use, but only up to a point. It's not shown on the graph, but that point is about 3 or 4 cores per socket. After that, the memory bandwidth is completely saturated, and adding cores does not improve performance. Performance at this plateau is less than 10% of peak.

In this collection of results, only local memory is used. Remote memory, or memory that is attached to the second NUMA domain, would be even slower to access.

PCHASE

PCHASE is a benchmark that measures unloaded memory latency. It gets its name because it creates a chain of randomly distributed pointers, then "chases" them repeatedly to measure the time it takes. The distribution of pointers is carefully designed to defeat any cache automatic prefetch mechanism in order to measure the native latency.

PCHASE allocates a pool of memory to be used for creating a chain of pointers. The size of that pool determines whether the chain fits into L1, L2, L3 cache, or main memory. The pool can also be allocated in local memory or remote memory, to determine those latencies as well. For these tests, only local memory was used.

A pointer chain is constructed from a pool of memory by first dividing the pool into cache lines. The cache lines are then arranged in a random order, to defeat any cache prefetch engine, and the pointer chain is constructed from the randomized cache lines. Exactly one pointer is allocated from each cache line. The benchmark allows the pointer to be allocated anywhere in the cache line, but for these tests, the pointer was always the first 8 bytes.

The benchmark is executed by starting at the beginning of the chain and following the chain of pointers to its end. This process is repeated until a specified number of iterations have completed. The elapsed time is known, as well as the length of the chain and the number of iterations that were completed. This information is used to compute the average latency of accessing a random cache line from the pool. The results for this benchmark are shown in Figure C.2.

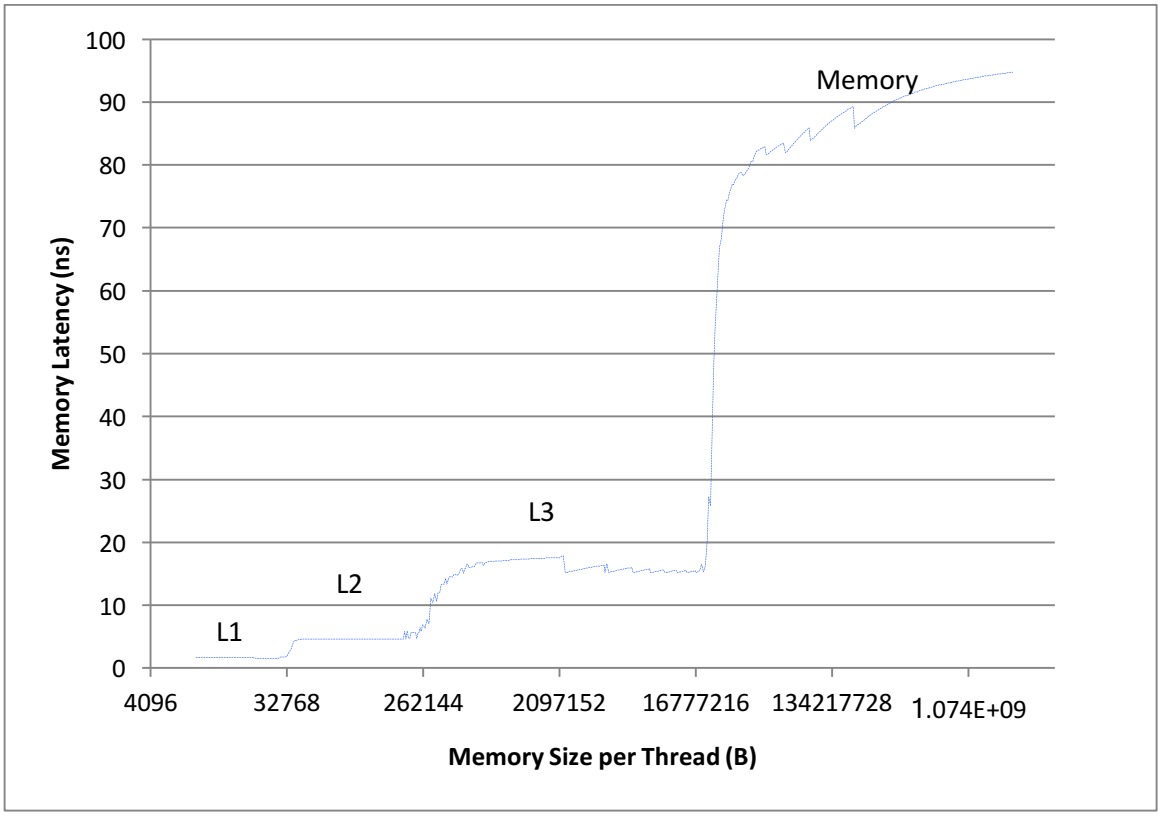

Figure C.2 – PCHASE Single Thread Unloaded Memory Latency

From the figure it is easy to see the random access latency of the E5-2670 memory hierarchy. L1 latency is from 3 to 5 clocks, L2 latency is 10 to 12 clocks, L3 latency is approximately 45 clocks, and local main memory is from 80 ns to 95 ns.

Even the split between near L3 and far L3 can be seen. This split comes about because each core has a block of 2.5 MiB of L3 cache. Each block is connected to every other block, creating a large shared L3 cache. In the chart, there is a transition in performance as the memory pool begins to spread into the L3 cache blocks of other cores. It is not clear to the author why this would be faster, but the transition is clearly visible.

The important characteristics of this benchmark are that each load operation fetches exactly one cache line, and the location of that cache line is known.

DGEMM

The DGEMM benchmark measures the performance of the DGEMM subroutine from the LAPACK math library. It calculates the product of two matrixes, plus the sum of a third. Specifically, it computes $\alpha AB + \beta C$, where α and β are scalars, and A, B and C are matrixes.

This benchmark has the characteristic of having very high floating-point computation rates, as seen in Figure C.3. It accomplishes this by moving data into cache and registers, then repeatedly

operating on that data while it is in registers. Because the data is in registers, its performance is much higher than can be achieved by DAXPY or other operations that operate on data in cache or memory. Furthermore, its performance is relatively insensitive to its location in memory.

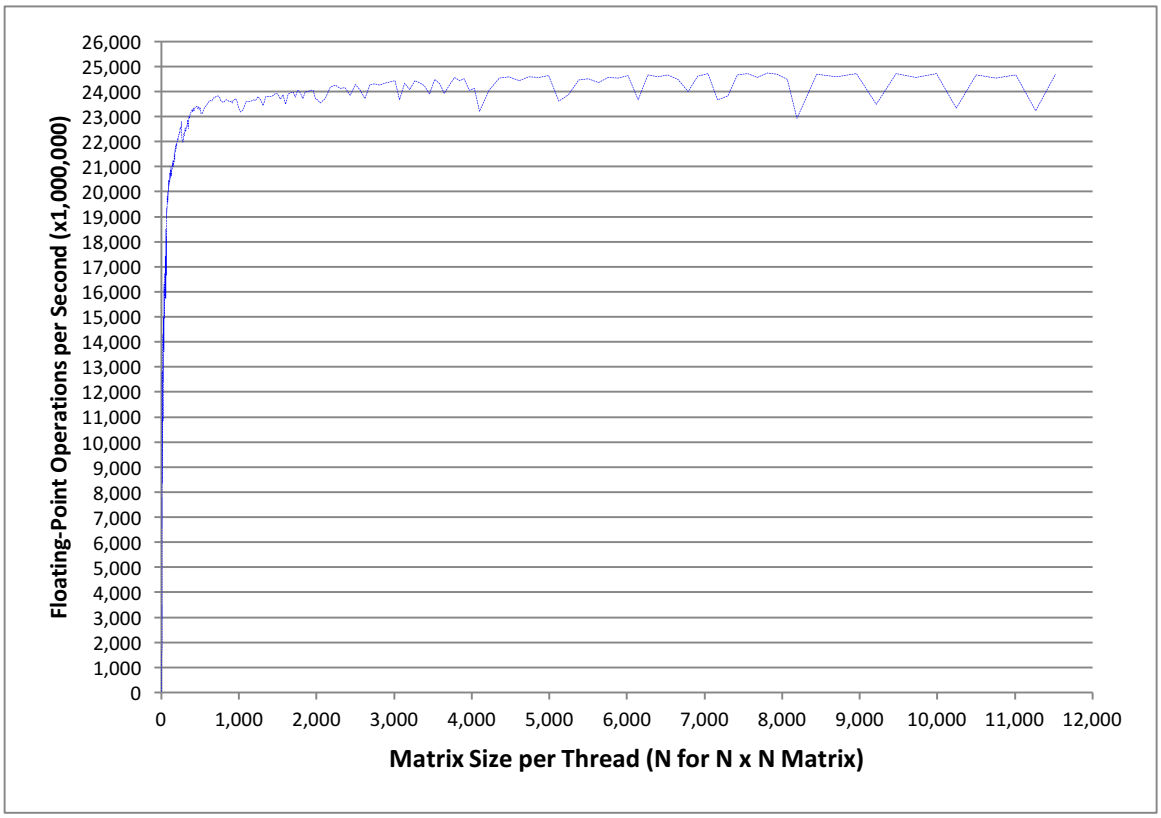

Figure C.3 – DGEMM Performance For 1 Core

The number of floating-point operations is also well known. For N x N matrixes A, B and C, the number of operations is $2N^3 + N^2$. The number of loads and stores is proportional to the number of floating-point operations, but the constant of proportionality is very small.

DGETRF

The DGETRF benchmark is essentially identical to the LINPACK benchmark. It solves a system of N linear equations in N unknowns. It accomplishes this by performing an LU factorization of an N x N double-precision floating-point matrix that represents the equation coefficients. LU factorization is well known to be a highly efficient solution to solving a system of equations in dense matrixes, which occurs often in the real world. The performance of this solution is shown in Figure C.4.

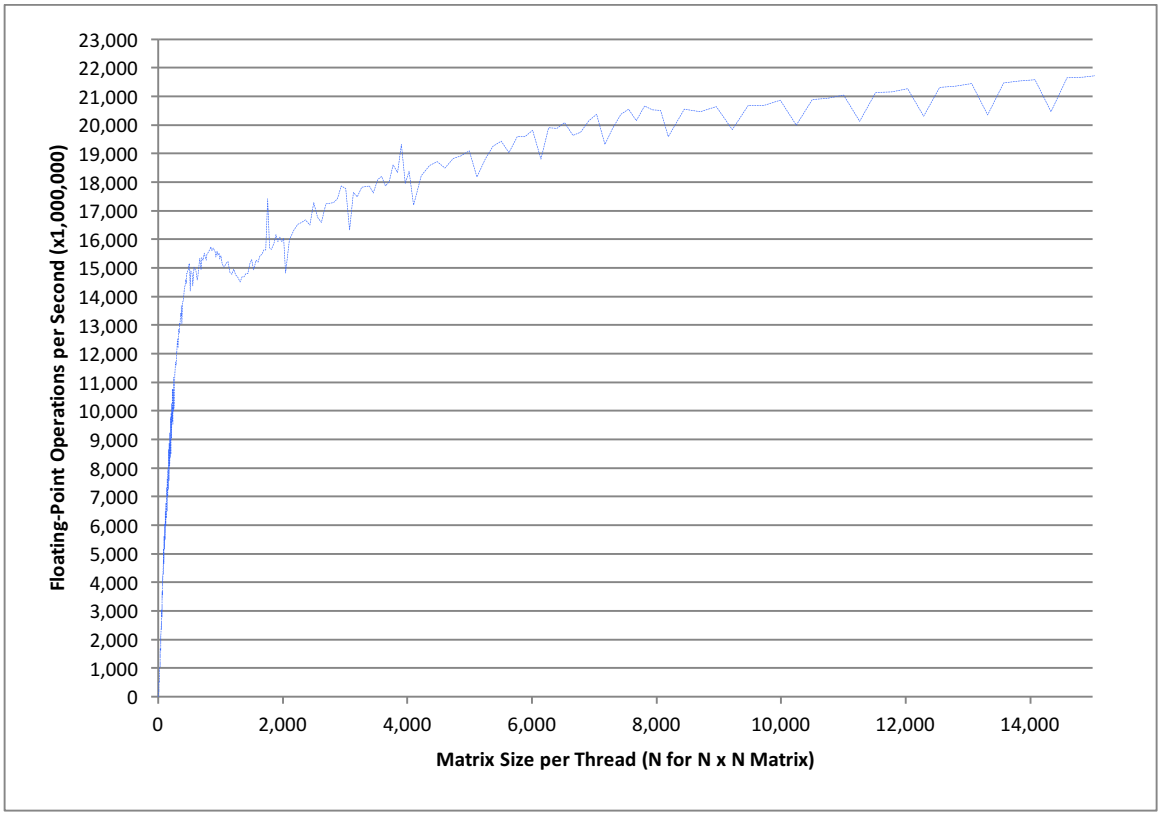

Figure C.4 – DGETRF Performance For 1 Core

The performance of LU factorization is well understood. The number of floating-point operations for LU factorization of an N x N matrix is $2N^3/3$. The number of operations for the back-fill portion, also included here, is $2N^2$. Like DGEMM, the number of load and store operations is proportional to the number of arithmetic operations and the constant of proportionality is very small. This is what gives both DGEMM and DGETRF a high degree of performance -- there are many operations for each load and store operation that references memory. It also makes them insensitive to the location of data in memory.

APPENDIX D - EVENT COUNTER VALUES

DAXPY

Event\Length	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
PAPI_REF_CYC	14,114,613,048	16,291,115,808	21,719,157,569	23,193,892,371	32,657,242,137	32,460,283,540	32,376,919,665
PAPI_REF_NS	5,441,884,247	6,281,029,288	8,373,813,925	8,942,395,442	12,590,986,536	12,515,049,334	12,482,908,643
PAPI_L1_TCM	53,265	2,505,015,106	2,521,161,599	2,523,099,867	2,521,814,545	2,520,013,217	2,521,670,685
PAPI_L2_TCM	963	59,735,431	1,493,143,231	1,522,354,584	1,765,581,926	1,774,159,718	1,777,214,102
PAPI_L3_TCM	2	1	56	40,466,928	305,232,575	307,974,306	309,284,335
PAPI_DP_OPS	20,000,157,921	22,893,727,521	24,133,977,879	24,132,340,591	24,183,706,030	24,219,081,919	24,178,759,265
PAPI_FDV_INS	81	115	108	108	172	117	130
PAPI_TOT_CYC	17,896,354,405	20,651,661,230	27,510,798,170	29,267,921,760	41,163,790,967	40,968,254,996	40,847,569,463
PAPI_TOT_INS	45,560,010,360	45,056,011,200	45,005,613,298	45,000,573,931	45,000,073,533	45,000,023,023	45,000,017,973
PAPI_VEC_DP	0	0	0	0	0	0	0

Table D.1 – PAPI Counter Values For Scalar DAXPY

Metric\Length	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
GF/s	3.6752	3.6449	2.8821	2.6986	1.9207	1.9352	1.9369
CPI	0.3928	0.4584	0.6113	0.6504	0.9147	0.9104	0.9077
Vectorization	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Ins/L1M	855,346	18	18	18	18	18	18
Ins/L2M	47,310,499	754	30	30	25	25	25
Ins/L3M	22,780,005,180	45,056,011,200	803,671,666	1,112	147	146	145
Turbo GHz	3.29	3.29	3.29	3.27	3.27	3.27	3.27
FP/Ins	0.4390	0.5081	0.5362	0.5363	0.5374	0.5382	0.5373
FP/L1M	375,484	9	10	10	10	10	10
FP/L2M	20,768,596	383	16	16	14	14	14
FP/L3M	10,000,078,961	22,893,727,521	430,963,891	596	79	79	78

Table D.2 – Derived Metrics For Scalar DAXPY

Event\Length	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
PAPI_REF_CYC	7,124,958,068	9,790,140,892	16,320,832,926	18,540,016,032	33,072,348,734	32,894,296,958	32,924,551,957
PAPI_REF_NS	2,747,025,004	3,774,585,419	6,292,490,250	7,148,095,290	12,751,030,749	12,682,383,178	12,694,048,282
PAPI_L1_TCM	20,782	2,505,019,320	2,534,489,633	2,532,714,504	2,517,494,384	2,516,230,630	2,516,200,806
PAPI_L2_TCM	856	916,435	2,450,046,098	2,286,454,309	2,285,801,777	2,297,223,312	2,299,880,139
PAPI_L3_TCM	8	16	115	84,091,006	812,224,757	855,456,676	855,181,170
PAPI_DP_OPS	20,000,139,760	25,678,267,355	30,031,582,803	29,980,127,191	29,823,973,087	29,932,718,543	29,920,737,429
PAPI_FDV_INS	43	56	69	61	114	129	1,749
PAPI_TOT_CYC	9,031,999,576	13,556,193,842	20,682,712,736	23,571,931,457	41,711,881,106	41,637,636,902	41,268,051,997
PAPI_TOT_INS	24,480,007,934	23,823,008,984	23,757,311,488	23,750,742,396	23,750,090,871	23,750,025,219	23,750,018,627
PAPI_VEC_DP	20,000,139,758	25,678,267,354	30,031,582,802	29,980,127,188	29,823,973,086	29,932,718,542	29,920,737,428

Table D.3 – PAPI Counter Values For Vector DAXPY

Metric\Length	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000	1,000,000,000
GF/s	7.2807	6.8029	4.7726	4.1941	2.3389	2.3602	2.3571
CPI	0.3690	0.5690	0.8706	0.9925	1.7563	1.7532	1.7376
Vectorization	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Ins/L1M	1,177,943	10	9	9	9	9	9
Ins/L2M	28,598,140	25,995	10	10	10	10	10
Ins/L3M	3,060,000,992	1,488,938,062	206,585,317	282	29	28	28
Turbo GHz	3.29	3.59	3.29	3.30	3.27	3.28	3.25
FP/Ins	0.8170	1.0779	1.2641	1.2623	1.2557	1.2603	1.2598
FP/L1M	962,378	10	12	12	12	12	12
FP/L2M	23,364,649	28,020	12	13	13	13	13
FP/L3M	2,500,017,470	1,604,891,710	261,144,198	357	37	35	35

Table D.4 – Derived Metrics For Vector DAXPY

PCHASE

Event\Size	12,800	128,000	1,280,000	12,800,000	128,000,000	1,280,000,000	12,800,000,000
PAPI_REF_CYC	6,500,990,068	19,046,656,307	70,998,805,505	101,454,405,051	498,164,028,229	525,288,765,337	784,743,520,704
PAPI_REF_NS	2,506,502,163	7,343,572,129	27,374,089,011	39,116,459,940	192,070,642,503	202,528,760,770	302,563,280,988
PAPI_L1_TCM	11,857	2,003,017,463	2,001,158,488	2,890,378,606	3,862,418,138	4,139,752,543	5,583,280,226
PAPI_L2_TCM	639	70,383	2,000,859,383	2,008,300,222	3,138,813,647	3,922,467,000	4,280,222,683
PAPI_L3_TCM	1	8	405	33,073,941	1,999,825,549	2,005,248,246	3,273,272,641
PAPI_DP_OPS	0	0	0	0	0	0	0
PAPI_FDV_INS	28	93	237	240	1,287	1,377	3,582
PAPI_TOT_CYC	8,241,872,681	24,048,734,497	89,913,133,879	127,301,912,164	630,259,469,026	665,507,200,992	993,918,026,631
PAPI_TOT_INS	6,110,007,670	6,011,012,479	6,001,132,531	6,000,153,918	6,000,211,485	6,000,209,069	6,000,307,938
PAPI_VEC_DP	0	0	0	0	0	0	0

Table D.5 – PAPI Counter Values For PCHASE

	12,800	128,000	1,280,000	12,800,000	128,000,000	1,280,000,000	12,800,000,000
CPI	1.3489	4.0008	14.9827	21.2164	105.0395	110.9140	165.6445
Ins/L1M	515,308	3	3	2	2	1	1
Ins/L2M	9,561,827	85,404	3	3	2	2	1
Ins/L3M	6,110,007,670	751,376,560	14,817,611	181	3	3	2
Turbo GHz	3.29	3.27	3.28	3.25	3.28	3.29	3.28

Table D.6 – Derived Metrics For PCHASE

DGEMM

Event\Size	500	1,000	1,500	2,000	2,500	3,000	10,000
PAPI_REF_CYC	31,951,495	228,364,091	741,147,781	1,731,110,625	3,348,056,733	5,757,587,042	209,928,892,934
PAPI_REF_NS	12,318,569	88,046,656	285,753,072	667,438,674	1,290,861,048	2,219,869,072	80,939,242,648
PAPI_L1_TCM	2,564,709	19,154,106	62,968,467	149,403,269	287,126,682	497,405,644	18,070,240,012
PAPI_L2_TCM	831,132	5,992,534	19,223,063	46,148,990	86,843,127	149,407,742	5,264,923,862
PAPI_L3_TCM	5,693	195,606	1,578,735	3,992,851	7,385,095	12,241,004	462,454,857
PAPI_DP_OPS	251,492,032	2,011,852,632	6,787,137,860	16,089,628,532	31,417,426,992	54,290,967,520	2,010,345,763,060
PAPI_FDV_INS	9,881	10,092	10,084	10,077	10,139	10,047	12,060
PAPI_TOT_CYC	38,506,140	285,317,428	933,803,606	2,187,699,592	4,235,645,463	7,285,678,107	265,830,802,688
PAPI_TOT_INS	101,050,933	782,921,812	2,631,250,594	6,223,028,795	12,146,819,377	20,971,679,570	775,658,321,113
PAPI_VEC_DP	251,492,032	2,011,852,632	6,787,137,860	16,089,628,532	31,417,426,992	54,290,967,520	2,010,345,763,060

Table D.7 – PAPI Counter Values For DGEMM

Metric\Size	500	1,000	1,500	2,000	2,500	3,000	10,000
GF/s	20.4157	22.8498	23.7518	24.1065	24.3383	24.4568	24.8377
CPI	0.3811	0.3644	0.3549	0.3515	0.3487	0.3474	0.3427
Vectorization	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Ins/L1M	39	41	42	42	42	42	43
Ins/L2M	122	131	137	135	140	140	147
Ins/L3M	17,750	4,003	1,667	1,559	1,645	1,713	1,677
Turbo GHz	3.13	3.24	3.27	3.28	3.28	3.28	3.28
FP/Ins	2.4888	2.5697	2.5794	2.5855	2.5865	2.5888	2.5918
FP/L1M	98	105	108	108	109	109	111
FP/L2M	303	336	353	349	362	363	382
FP/L3M	44,176	10,285	4,299	4,030	4,254	4,435	4,347

Table D.8 – Derived Metrics For DGEMM

DGETRF

Event\Size	500	1,000	2,000	4,000	8,000	12,000	16,000
PAPI_REF_CYC	27,559,904	130,751,535	833,977,269	5,767,280,067	41,870,078,689	141,492,512,315	317,801,419,593
PAPI_REF_NS	10,626,154	50,412,481	321,545,264	2,223,607,429	16,143,238,373	54,553,195,959	122,527,574,663
PAPI_L1_TCM	1,509,624	9,138,055	60,238,941	442,449,048	3,249,003,472	10,953,800,961	25,540,010,668
PAPI_L2_TCM	556,867	3,053,019	19,168,881	130,682,493	910,716,259	3,183,829,111	6,985,802,798
PAPI_L3_TCM	48,278	256,991	2,004,120	13,426,458	106,264,626	355,758,057	761,798,778
PAPI_DP_OPS	85,645,277	676,994,679	5,384,406,358	42,947,417,510	342,885,907,317	1,156,586,098,579	2,740,391,167,242
PAPI_FDV_INS	12,091	14,170	18,132	26,291	45,011	67,357	92,812
PAPI_TOT_CYC	26,730,419	146,917,371	1,007,833,659	7,151,654,319	51,750,061,386	176,360,551,759	397,279,902,727
PAPI_TOT_INS	45,036,478	299,122,557	2,227,085,492	17,199,520,323	134,984,713,172	452,878,984,034	1,069,849,352,156
PAPI_VEC_DP	85,636,586	676,976,188	5,384,369,460	42,947,341,068	342,885,749,300	1,156,585,838,866	2,740,390,825,276

Table D.9 – PAPI Counter Values For DGETRF

Metric\Size	500	1,000	2,000	4,000	8,000	12,000	16,000
GF/s	8.0599	13.4291	16.7454	19.3143	21.2402	21.2011	22.3655
CPI	0.5935	0.4912	0.4525	0.4158	0.3834	0.3894	0.3713
Vectorization	0.9999	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
Ins/L1M	30	33	37	39	42	41	42
Ins/L2M	81	98	116	132	148	142	153
Ins/L3M	933	1,164	1,111	1,281	1,270	1,273	1,404
Turbo GHz	2.52	2.91	3.13	3.22	3.21	3.23	3.24
FP/Ins	1.9017	2.2633	2.4177	2.4970	2.5402	2.5539	2.5615
FP/L1M	57	74	89	97	106	106	107
FP/L2M	154	222	281	329	377	363	392
FP/L3M	1,774	2,634	2,687	3,199	3,227	3,251	3,597

Table D.10 – Derived Metrics For DGETRF

DISTRIBUTION

1 KRELL Institute
Attn: Don Maghrak
1609 Golden Aspen Drive, Suite 101
Ames, IA 50010

1	MS0807	Anthony M. Agelastos	9326
1	MS0807	Douglas M. Pase	9326
1	MS0807	Joel Stevenson	9326
1	MS0823	Constantine Pavlakos	9326
1	MS0931	Steven E. Thomas	9322
1	MS0899	Technical Library	9536 (electronic copy)


Sandia National Laboratories