LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-406864

Measuring FLOPS Using Hardware Performance Counter Technologies on LC systems

Dong H. Ahn

September 9, 2008

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Measuring FLOPS Using Hardware Performance Counter Technologies on LC systems

Dong H. Ahn
Livermore Computing,
Lawrence Livermore National Laboratory
ahn1@llnl.gov

Abstract: *FLOPS (FLoating-point Operations Per Second) is a commonly used performance metric for scientific programs that rely heavily on floating-point (FP) calculations. The metric is based on the number of FP operations rather than instructions, thereby facilitating a fair comparison between different machines. A well-known use of this metric is the LINPACK benchmark that is used to generate the Top500 list. It measures how fast a computer solves a dense N by N system of linear equations Ax=b, which requires a known number of FP operations, and reports the result in millions of FP operations per second (MFLOPS). While running a benchmark with known FP workloads can provide insightful information about the efficiency of a machine's FP pipelines in relation to other machines, measuring FLOPS of an arbitrary scientific application in a platform-independent manner is nontrivial.*

The goal of this paper is twofold. First, we explore the FP microarchitectures of key processors that are underpinning the LC machines. Second, we present the hardware performance monitoring counter-based measurement techniques that a user can use to get the native FLOPS of his or her program, which are practical solutions readily available on LC platforms. By nature, however, these native FLOPS metrics are not directly comparable across different machines mainly because FP operations are not consistent across microarchitectures. Thus, the first goal of this paper represents the base reference by which a user can interpret the measured FLOPS more judiciously.

1 Floating Point Pipeline

1.1 AMD Dual-Core and Quad-Core Opteron: Peloton and TLCC

Over the last two decades, the x86 instruction set architecture and its 64-bit extension, x86-64, have added various support to improve their FP performance. Intel's 80486 was the first x86 CPU that integrated the x87 FP unit into the chip. Then, its Pentium line added the MMX instruction set that enabled a form of SIMD FP operations. In response, AMD added the 3DNow! instruction set extension starting from its K6-2 CPU, further enhancing MMX. Intel subsequently improved the concept of SIMD operations with its SSE (Streaming SIMD Extension) instruction set extension into Pentium III. Ever since, Intel and AMD have continued to enhance the SSE-based technologies, adding incremental upgrades to the base SSE technology (i.e., SSE2, SSE3, SSE4, and etc.).

The microarchitectures supporting these x87 instructions and other extensions vary from one CPU to another. Generally, however, the x87 FP instructions mostly flow through an 80-bit scalar FP pipeline that operates on a FP register stack consisting of eight 80-bit registers, and the MMX instructions operate on MMn registers that were originally just aliases to the x87 FP unit stack registers, but later mapped to more efficient register files. A series of SSE instruction set extensions has been the response to the shortcomings of initial SIMD instruction set extensions such as the MMX extension. SSE instructions are designed to operate on 128-bit vector registers named XMM registers, making it easy to perform SIMD and scalar x87 FP operations simultaneously.

In the case of Opteron x86-64 processors, AMD's out-of-order, superscalar 64-bit processors that are compatible with the x86 ISA, these SSE extensions operate on sixteen 128-bit XMM registers. Hence, they are capable of executing up to two packed double-precision FP operations or four packed single-precision FP operations with a single instruction. However, the Dual-Core Opteron implementation does not have a wide enough data path that enables

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-TR-406864).

an instruction to execute the entire 128-bit wide FP operations at once. Thus, it chops the instruction into two 64-bit micro-operations (*uops*) and executes each *uop* one after another: this processor requires two cycles to complete two double-precision FP operations through its SSE extensions. To overcome this limitation, the Quad-Core Opteron implementation widened the data path such that its SSE execution pipeline can perform two double-precision FP operations per cycle without having to split an instruction into two *uops*.

LC's recent Linux clusters are based on processors in this AMD Opteron family: the Dual-Core Opteron Processor 8216 powers LC's Peloton class machines such as `Atlas`, `Zeus`, `Rhea`, `Hopi` and `Minos` while the Quad-Core Opteron Processor 8354 are the microprocessor of choice for TLCC class clusters such as `Juno` and `Hype`.

Due in large part to pressure on backward compatibility, an x86 or x86-64 family processor must support most of these instruction set extensions. Clearly, collecting an accurate FLOPS through these multiple pipelines is a daunting task as a user must profile dynamic events occurring over all the FP execution pipelines.

1.2 IBM Power4 and Power5[+]: Purple

The POWER5 (and POWER5+) processor chip contains two 64-bit POWER5[+] cores. Identical to a POWER4 core, this out-of-order, superscalar POWER5 core contains two FP execution units, each of which forms an independent FP pipeline. Significantly improved from their predecessor chips (i.e., POWER3), these FP units support six-staged execution pipelines for all PowerPC FP instructions including the FP divide instruction and the FP fused multiply-add instruction that makes the theoretical peak of a core four FP operations per cycle. On POWER3, the FP divide and the FP square root instructions are not pipelined, hindering certain applications from getting optimal performance. These Power4 and Power5 are the microprocessors on Purple machines. `Um` and `Uv` that had been delivered earlier consists of Power4 while `Purple` and `Up`, newer Purple machines, are equipped with Power5+.

1.3 IBM PowerPC 440 with the Double Hummer FP Unit: BlueGene/L

The IBM PowerPC 440 embedded microprocessor that embodies BlueGene/L implements a dual-issue FP unit. This unit consists of two sets of FP register files, the primary file (32 64-bit FP registers) and the secondary file (32 64-bit FP registers), capable of executing the original PowerPC Book E FP instruction set and the SIMD-like Oedipus instruction set. Oedipus is an instruction set extension designed to boost FP complex arithmetic and other FP SIMD operations. The secondary register file forms a true FP register file that is used to hold additional operands for these new instructions. Given each processor running at 700 MHz and the FP unit supporting two packed double-precision fused multiply-add instructions, the theoretical peak of each PowerPC core is 2.8 GFLOPS.

2 Hardware Counter Technologies and Native FP Operation Events

Modern microprocessors include integrated hardware support for non-intrusive monitoring of a variety of processor and memory subsystem events. Commonly referred to as hardware counters, this capability is very useful to both computer architects and application developers. Detailed software instrumentation can introduce perturbation into an application and the measurement process itself. On the other hand, simulation can become impractical for large, complex applications. These hardware performance monitoring counters fill a gap that lies between detailed microprocessor simulation and software instrumentation because they have relatively low perturbation and can provide an insight into the processor and memory-system behavior [3].

These hardware counters are typically implemented as a small set of registers embedded into a microprocessor, counting events, occurrences of specific signals related to the processor's function. In particular, counters of most contemporary microprocessors can monitor various signals that arise as a result of dynamic FP instructions flowing through FP pipelines. For example, IBM POWER5 can count various FP instructions executed through each of two FP units (i.e., the number of FP divide instructions executed through the first FP unit, the number of FP fused multiply-add instructions through the second unit, etc.), and also count other events like a FP unit receiving a denormalized operand.

The granularity and event types that hardware counters can measure are specific to a processor. Even within revisions of the same microprocessor family, hardware counter architectures and capabilities can differ widely. In the following, we detail hardware counter architectures of various LC platforms, and hardware events relevant to measuring FLOPS.

2.1 Hardware Counters in the AMD Opteron Microprocessor Family

Both AMD Dual-Core and Quad-Core Opteron processors have four integrated 48-bit hardware counter registers, each of which is associated with a dedicated event selection register. Tools or user codes can write bit fields into an event selection register to count a specific event through its associating hardware counter. The main differences between Dual-Core and Quad-Core lie in the use of event selection registers, as Quad-Core uses all 64 bits of each selection register, while Dual-Core only uses the lower 32 bits, and in available event types and their semantics.

2.1.1 FP Operation Events for AMD Dual-Core Opteron

On an AMD Dual-Core processor, we consider two hardware events to measure FLOPS: the `Retired FP/MMX Instructions` (RFI) event and the `Dispatched FPU Operations` (DFO) event [1]. RFI counts all retired FP operations executed through x87, MMX, 3DNow!, and SSE-based instructions, excluding those operations speculatively executed but never completed. Unfortunately, this metric contains FP loads and stores in addition to FP computations, thereby resulting in counts that are consistently significantly higher than the expected FLOPS, often by factors of two or more. Hardware counters on this processor provide an opportunity to correct this as a hardware event can measure FP loads and stores. However, the correction factors themselves are speculative, and can easily lead to undercounting errors similar in magnitude to those seen in the pure speculative counts.

On the other hand, the DFO event counts only FP computations. However, it includes those speculative operations that never graduate from the system, leading to over-counting errors. The margin of error can be significantly higher on complex production codes.

We also note that on this CISC processor these counts are based on the number of *uops* rather than original instructions. Internal to an x86 processor, *uops* form the basic micro-operation set in which one or more *uops* are mapped to a high level instruction visible to programmers. Particularly, such an architecture maps some of more complex FP instructions like divide and square root into a combination of multiple FP *uops*, which becomes another source for over-counting errors [7, 5]. For example, our measurements indicate that the dynamic instruction mix of an LLNL multi-physics code contains four percent FP divide and square root instructions, each of which is measured to require around five *uops* on this processor.

Taken together, our recommendation for this processor is to use the DFO metric as the upper bound for FLOPS measured at the *uop* level. But, should the lower bound of FLOPS be needed, a user should consider using RFI corrected with the speculative FP load and store instruction count.

2.1.2 FP Operation Events for AMD Quad-Core Opteron

Besides the RFI and DFO metrics, whose semantics remains identical, AMD added the `Retired SSE Operations` (RSO) event to the Quad-Core processor, a new event designed to count all retired SSE instructions precisely [2]. The event can be configured to measure the number of FP operations either directly at the *uop* level or at the FLOP level, as AMD defines FLOP based on their proprietary information. Clearly, RSO does not capture x87, or MMX and 3DNow! instructions, but can allow collecting of precise data points at the FLOP level for all SSE instructions. Moreover, AMD began deprecating support for some of the old FP pipelines such as x87 for its 64-bit computing; thus this event can capture the lion's share of FP operations.

Taken together, our recommendation for this processor is still to use DFO as the upper bound for FLOPS, which provides a metric at the *uop* level directly comparable to the Dual-Core processor. Additionally, we recommend measuring RSO both at the *uop* level and the FLOP level, which can provide an additional insight into more accurate FP computation intensity for SSE operations.

2.2 Hardware Counters in the IBM Power4/Power5[+] Microprocessor Family

Both Power4 and Power5 cores have a limited set of integrated performance registers to monitor hardware events. Power4 contains eight counters, and Power5 extends the capability to the thread level in order to support Simultaneous Multi-Threading (SMT), dedicating six counters per each SMT thread. Counters can be independently configured to monitor more than 300 performance events occurring on the processor or memory subsystem.

To measure an accurate FLOPS count on Power4 and Power5 cores, we must consider three metrics: the `PM_FPU_FIN` (FPU produced a result) event, the `PM_FPU_STF` (FPU executed store instruction) event, and the `PM_FPU_FMA` (FPU executed multiply-add instruction) event. `PM_FPU_FIN` counts the combined number of cycles in which each of two FP

units produces a result. But this event includes not only all of the numeric instructions but also FP stores (but not FP loads). In addition, it only counts a FP fused multiply-add instruction as a single operation despite the fact that this instruction normally executes two FP operations. `PM_FPU_STF` and `PM_FPU_FMA` can be measured to correct these factors:

$$FLOPcount = PM\_FPU\_FIN - PM\_FPU\_STF + PM\_FPU\_FMA \quad (1)$$

In (1), subtracting PM_FPU_STF corrects the count for FP stores, and adding PM_FPU_FMA gives a weight to fused multiply-add instructions twice as much as other scalar floating instructions.

Thus, our recommendation for these processors is to normalize this derived metric against the execution time, and to use it as the upper bound for FLOPS. This metric is an upper bound as a compiler can use fused multiply-add instructions in place for simple add (`fadd`) or multiply (`fmul`) by applying zero to an operand to cancel out either operation. Furthermore, all three events used in the metric include uncommitted speculative instructions.

2.3 Hardware Counters in the IBM PowerPC 440d Embedded Processor

The BlueGene/L compute node contains a large number of performance counters a.k.a the Universal Performance Counters in both of its PowerPC 440d embedded processor cores as well as interconnects, the torus and tree network alike [6]. The system maps these counters in the Device Control Bus, thereby allowing the access through Device Control Registers (DCR). Similar to other microprocessors, this system has much larger set of hardware events than there are performance counters, and allows simultaneous monitoring of up to 48 events.

As for events germane to FLOPS, unfortunately, the hardware counter facility does not include events for measuring SIMD add/subtract or multiply instruction counts, making a profiling cumbersome. A user must obtain relevant information from two separate measurements of different builds. As decribed in [4], the steps are

1) Compile the code without SIMD instructions (i.e., use -qarch=440 with the xlC compiler), using unoptimized (non-SIMD) versions of computationally intense libraries. Measure the total FPU operation count with this executable. 2) Recompile the code, enabling the SIMD instructions (using -qarch=440d). Obtain the total number of cycles and, thus, the total time with this executable. The FP operation count in this case is potentially inaccurate and should be discarded. 3) Divide the total FP operation count by the total time to compute the performance.

3 Software Tools, Libraries, and Techniques

3.1 Performance Application Programming Interface

PAPI is a popular software specification of a cross-platform interface to hardware performance counters. In addition to the specification, the PAPI project has developed a reference implementation of this specification as a library that has been widely deployed on HPC systems. This implementation uses a layered approach where its machine-dependent layer hides underlying native interfaces and hardware details from its the upper layer, enhancing portability. For each platform, this reference implementation attempts to map as many of the PAPI standard events as possible to native events on that platform.

The most relevant PAPI standard events germane to FLOPS are `PAPI_FP_OPS` and `PAPI_FP_INS`. On Dual-Core Opteron, the `PAPI_FP_OPS` event is currently mapped to DFO and the `PAPI_FP_INS` event is to RFI while on Quad-Core both `PAPI_FP_OPS` and `PAPI_FP_INS` are mapped to RSO in FLOP mode (see section 2.1). A user can bracket his code using interface calls programmed with these events, and get native FLOPS over a section of the code.

3.1.1 PapiEx

Although measuring FLOPS over a key section of the code is useful, users often have to measure FLOPS over the entire execution without having to instrument the code. `papiex` is a performance analysis tool designed to assist a user with such an effort. It transparently and passively measures the hardware performance counters of an application using the PAPI library. Furthermore, it intercepts creation and destruction of processes and threads, thereby reporting performance monitoring information for all threads and child processes of a program. However, it is not a tool for selective instrumentation.

3.1.2 IBM hpmcount

`hpmcount` is an IBM utility that starts an application and provides at the end of execution wall clock time, hardware performance counters information, derived hardware metrics, and resource utilization statistics. It used to be a component of HPM Toolkit under IBM Alphaworks, but since became part of the base AIX OS. The utility command is typically available in /usr/pmapi/tools on AIX.

`hpmcount` also comes with an instrumentation library (libhpm) that provides an instrumented program with a summary output for each instrumented region in the program. This library supports serial and parallel (MPI, threaded, and mixed mode) applications, written in Fortran, C, and C++.

On Power4 and Power5 systems, this utility uses the native Performance Monitoring API (PMAPI) that restricts accessing of individual events. Thus, an event must be accessed using a group ID through a predefined PMAPI event group to which a target event belongs. For example, the `pm_fpuX5` (Floating point and L1 events) group collects `PM_FPU0_FIN` and `PM_FPU1_FIN` whose addition is equal to `PM_FPU_FIN`, and also reports `PM_FPU_STF`. On the other hand, the `pm_fpuX4` (Floating point events) group is one of the groups that allows measuring of `PM_FPU_FMA`.

4 Putting it all together

In this section, we measure the native FLOPS of a simple code on various LC platforms using the software tools and techniques described.

4.1 DAXPY

We use as our test program a Double-precision scalar Alpha X Plus Y, DAXPY, which is a common operation used in scientific applications. It is a combination of scalar multiplication and vector addition:

$Y = \alpha \cdot X + Y$ where α is a scalar, and X and Y are vectors.

Listing 1: DAXPY.c

```
/*
 * Expected FLOP Count: 2 * SIZE (SIZE is defined below).
 *
 */
#define SIZE 100000

double X[SIZE];
double Y[SIZE];
const double PI    = 3.141592;
const double alpha = -2.30459;

int main ( )
{
  int i;

  for (i=0; i < SIZE; ++i) {
    X[i] = PI;
    Y[i] = PI;
  }

  for (i=0; i < SIZE; ++i)
    Y[i] = alpha * X[i] + Y[i];

  return 0;
}
```

4.2 Measuring FLOPS Using hpmcount on IBM Power5+: the Up Machine

On the `Up` machine, we first compile DAXPY.c using the xlc compiler with no optimization flags.

```
up041{user}185:  xlc -O0 DAXPY.c -o DAXPY
```

We then create the hpmcount input text file, `libHPM_events`, in the directory where we run our experiments, and type `pm_fpuX4` into this file. Next, we type

```
up041{user}186:  hpmcount -o hpmout.fpuX4 DAXPY
```

At the end of the execution, performance counter data are dumped into an output file whose file name is prefixed with hpmout.fpuX4. The output file reports the metrics in its hardware counter section as shown in Listing 2.

Listing 2: pm_fpuX4

```
PM_FPU_SINGLE (FPU executed single precision instruction) :          0
PM_FPU_STF (FPU executed store instruction)               :     306500
PM_FPU0_FIN (FPU0 produced a result)                      :     306222
PM_FPU1_FIN (FPU1 produced a result)                      :     100195
PM_INST_CMPL (Instructions completed)                     :    4692373
PM_RUN_CYC (Run cycles)                                   :    5324768
```

To collect PM_FPU_FMA, we then replace `pm_fpuX4` with `pm_fpuX5` in the `libHPM_events` file, and run another hpmcount experiment. Listing 3 shows the PM_FPU_FMA count.

Listing 3: pm_fpuX5

```
PM_FPU_FMA (FPU executed multiply-add instruction)          :          100012
```

Using the derived metric shown in (1), FLOPcount then becomes 199,929 (306,222 + 100,195 - 306,500 + 100,012). Notice that this value is very close to the expected FLOP count: 200,000; during an execution, 100,000 fused multiply-add instructions would have been completed to update 100,000 elements with the DAXPY operation. Dividing this by the execution time, FLOPS works out to be 32 MFLOPS.

4.3 Measuring FLOPS Using PapiEx on AMD Dual-Core Opteron: the Atlas Machine

Similarly on the `Atlas` machine, we first compile DAXPY.c using the Intel compiler with no optimization flags

```
atlas544{user}31:  icc -O0 DAXPY.c -o DAXPY
```

Because `PAPI_FP_OPS` is configured with our recommended metric on this microprocessor, we simply run DAXPY under the control of `papiex` utility with `PAPI_FP_OPS`.

```
atlas544{user}32:  papiex -e PAPI_FP_OPS ./DAXPY
```

The `papiex` output is dumped into a file named target_code.papiex.machine.pid by default, and hence the experiment creates DAXPY.papiex.atlas544.34314. Listing 4 shows the snippet of this file.

Listing 4: DAXPY.papiex.atlas544

```
Proc usecs:                    3252
Proc cycles:                   7844059
I/O cycles:                    0
PAPI_FP_OPS:                   200018
```

FLOPcount is 200,018, and FLOPS works out to be 61 MFLOPS.

4.4 Measuring FLOPS Using PapiEx on AMD Quad-Core Opteron: the Juno Machine

On the `Juno` machine, we also compile DAXPY.c using the Intel compiler with no optimization flags. But on this microprocessor because `PAPI_FP_OPS` is not configured with our recommended metric (DFO), but rather RSO, we run DAXPY under the control of `papiex` configured with native DFO metrics: `DISPATCHED_FPU:OPS_MULTIPLY` and `DISPATCHED_FPU:OPS_ADD`. The native `DISPATCHED_FPU:OPS_MULTIPLY` event measures the number of *uops* executed through the FP multiply pipeline, and `DISPATCHED_FPU:OPS_ADD` through the FP add pipeline. These two execution pipelines form the lowest unit that serves all FP numeric computations on this computer architecture.

```
juno0{user}112:  papiex -e DISPATCHED_FPU:OPS_MULTIPLY -e DISPATCHED_FPU:OPS_ADD ./DAXPY
```

Listing 5 shows the snippet of the resulting output file.

Listing 5: DAXPY.papiex.juno0

```
Proc usecs:                    13472
Proc cycles:                   14814952
I/O cycles:                    0
DISPATCHED_FPU:OPS_MULTIPLY:   100016
```

DISPATCHED_FPU : OPS_ADD :	100017

FLOPcount is 200,033, and FLOPS works out to be 14 MFLOPS.

References

[1] Bios and kernel developer's guide for amd athlon 64 amd opteron processors, Febrary 2006.

[2] Bios and kernel developer's guide (bkdg) for amd family 10h processors, March 2008.

[3] D. H. Ahn and J. S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16. IEEE Computer Society Press, 2002.

[4] F. Gygi, E. W. Draeger, M. Schulz, B. R. de Supinski, J. A. Gunnels, V. Austel, J. Sexton, F. Franchetti, S. Kral, C. W. Ueberhuber, and J. Lorenz. Large-scale electronic structure calculations of high-Z metals on the bluegene/L platform. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 45. ACM Press, 2006.

[5] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, Mar./Apr. 2003.

[6] P. Mindlin, J. R. Brunheroto, L. D. Rose, and J. E. Moreira. Obtaining hardware performance metrics for the bluegene/L supercomputer. In *Proceedings of the 9th International Euro-Par Conference*, volume 2790 of *Lecture Notes in Computer Science*, pages 109–118, Klagenfurt, Austria, Aug. 2003. Springer-Verlag (Berlin/New York).

[7] S. F. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 14–16. IEEE Computer Society Press, 1999.