The trend in high-performance computing is to include computational accelerators such as GPUs or Xeon Phis in each node of a large-scale system. Qualitatively, such accelerators tend to favor codes that perform large numbers of floating-point and integer operations per branch; that exhibit high degrees of memory locality; and that are highly data-parallel. The question we address in this work is how to quantify those characteristics. To that end we developed an application-characterization tool called Byfl that provides a set of “software performance counters”. These are analogous to the hardware performance counters provided by most modern processors but are implemented via code instrumentation—the equivalent of adding \( \text{flops} = \text{flops} + 1 \) after every floating-point operation but in fact implemented by modifying the compiler’s internal representation of the code.

The novelty of our approach is that we report counter values in a hardware-independent manner. Unlike similar tools based on, say, PAPI [1] or Pin [2], Byfl (a) reports the same data regardless of target platform and (b) operates on a machine abstraction that is closer to the application developer’s view of the application than the hardware’s. For example, if a program includes a 64-bit scalar floating-point division, Byfl reports that as such, even if, as in Intel’s Knight’s Corner, this division is implemented in terms of dozens of primitive floating-point and integer instructions, mostly vector operations. Also, unlike static analysis tools such as ROSE [3], Byfl can handle control flow and data dependencies that are not known until run time.

Byfl is not constrained to reporting only what hardware performance counters might report. Two metrics that Byfl reports that are worthy of additional mention are unique bytes and flop (or op) bits. Unique bytes represent the program’s memory footprint—the number of unique, byte-level addresses that the program accessed during a run as opposed to the total number of bytes loaded and stored, often repeatedly. Flop bits is a creation of ours that attempts to rationalize the oft-quoted byte:flop ratio of an application. Consider the assignment “\( A \leftarrow B + C \)”, where each variable resides in memory. If the variables are all 32 bits wide, this assignment has a byte:flop ratio of 12. If, however, they are 64 bits wide, this same assignment has a byte:flop ratio of 24. In both cases, though, all data loaded from memory a fed into a floating-point operation, and all results of a floating-point operation are written to memory. The bits:flop-bit ratio normalizes this case to 1 by defining a flop bit as the number of bits consumed or produced by a floating-point operation (e.g., 192 for a 64-bit binary operator or 64 for a 32-bit unary operator). Bit:flop-bit values less than 1 imply register reuse, and values greater than 1 imply data traffic that is not bound to floating-point operations. Op bits are the analogue to flop bits for all arithmetic and logical operations, not just floating-point.

Table 1 presents the result of instrumenting the first process of a 1056-process run of the xRAGE radiation-hydrodynamics application [4] simulating an asteroid impact. The left half of Table I represents raw data, and the right half represents ratios of various raw-data values. As the data show, the instrumented process performed a total of 82 billion operations of which 5 billion represented floating-point operations. “Bad” branches—those with statically unpredictable target addresses—also numbered 5 billion. Hence, one can conclude that xRAGE is a somewhat control-intensive application, as an average of only 15 operations (< 1 floating-point operation) occur between potential pipeline flushes. xRAGE can therefore be expected to be challenging to optimize for most accelerators, which favor large stretches of branchless execution. On the other hand, the xRAGE process performed 140 billion bytes of memory accesses representing only 568 million distinct byte addresses. In other words, each byte of data was reused an average of 246 times. This is an encouraging metric from the perspective of creating a bit:flop-bit ratio that is more than 1 but less than 246, which is more realistic than the simplistic value of 246.6592 bytes per unique byte.
of minimizing data transfers between CPUs and accelerators.

Table I also shows bytes per flop. The xRAGE process consumed over 27 bytes of memory for every floating-point operation it performed. Examining instead the number of bits per flop bit we see a ratio of 1.2, implying that more data were loaded/stored to/from main memory than were consumed/generated by a floating-point unit. While these values sound pessimistic, being beyond the capabilities of modern memory systems, the data also show that the xRAGE process consumed 1.7 bytes per arbitrary (not necessarily floating-point) operation or, arguably more meaningfully, 0.15 bits per operation bit. That is, the ALU operated on more than six bits for each bit loaded or stored, implying that register usage is approximately six times as prevalent as memory usage—not much but still less worrisome than the initial 27 bytes/flop metric. (Also, caches and prefetchers may alleviate some of the bandwidth requirement.) Furthermore, the data show that xRAGE needs only 0.0069 unique bytes from memory for every operation it performs. That is, if xRAGE could fit its entire working set in cache it would be able to perform 145 operations for every mandatory (“cold”) cache miss, a far more optimistic characterization of the application’s memory-performance needs than the venerable bytes-per-flop metric indicates.

Because Byfl can gather data on a per-basic-block level we can examine the variability in computational intensity across an entire run. Figure 1 presents a box-and-whiskers plot of the flop-bits/bit ratio for three applications (xRAGE [4], Chicoma [5], and S3D [6]) and the SPEC CPU2006 floating-point benchmarks [7]. The center line of each box represents the median flop-bits/bit ratio across all basic blocks. The top and bottom of each box represent, respectively, the 75th and 25th percentiles of the data. The upper and lower whiskers represent ±1.5 interquartile ranges (IQRs) of the box boundaries. Outliers of the whisker boundaries are plotted individually as circles. The horizontal line at y = 1 separates floating-point-intensive points from memory-intensive points; higher points are likely to be better for accelerators than lower points.

Immediately apparent in Figure 1 is the abundance of outliers. This implies that constructing hardware requirements around an average computational intensity fails to take into consideration that some parts of an application are likely to be extremely memory-bound and will observe bad performance even on hardware with the “ideal” memory bandwidth based on the application’s average. Second, we can use a graph like Figure 1 to draw analogies between applications and benchmarks: both Chicoma and GemsFDTD exhibit a median computational intensity of zero; both xRAGE and dealII load or store approximately one memory bit for every bit produced or consumed by a floating-point unit; and both S3D and tonto compute slightly more than they access memory and see similar variance across basic blocks. All three applications see far more outliers than their corresponding benchmark, however.

The conclusions one can draw from this work are that (1) hardware-independent application characterization can be a useful counterpart to traditional hardware performance counters, binary modification, and source-to-source translation, (2) measuring compute intensity in terms of bits rather than bytes can reduce the misleading effect of different word widths and operand counts, (3) software performance counters enable applications to be evaluated for suitability to emerging architectures, and (4) applications tend to have more variability in compute intensity over time than do benchmarks, even those with similar average compute intensity.

Byfl is available from https://github.com/losalamos/Byfl.

REFERENCES


