Single-Core Performance is Still Relevant in the Multi-Core Era

Todd Mytkowicz
Microsoft Research
toddm@microsoft.com

Mark Marron
IMDEA Software Institute
mark.marron@imdea.org

Problem: Underutilized Hardware. Software rarely uses all the potential performance available in a modern microprocessor. For example, on an Intel Core 2 class workstation—a microprocessor capable of executing 4 instructions per cycle—the average instructions per cycle for the single-threaded DaCapo benchmark suite is 0.98. In other words, even in the multi-core era, there is still enormous potential to increase *single-core* program performance. Recent work in PLDI has focused on multi-core performance: by our count PLDI'10 has around 6 papers on multi-core optimizations and PLDI'11 has 6 papers as well.

We believe this aggressive focus on multi-core may miss a critical trend in computing environments: power, or performance per watt. The issues of power consumption and thermal dissipation are now major limiting factors in performance, even in environments with unconstrained power and cooling systems (e.g., desktops or servers). However, power consumption and performance per watt are even more critical in mobile computing (e.g., phones or tablets) and data centers, which are increasingly important computing environments. We argue that, while research on multi-core optimizations is valuable, improvements in single-core performance via improved resource utilization are key to increasing performance and doing so with minimal impact on power consumption.

Increasing Performance Per Watt via SIMD. In order to understand the performance and efficiency space better we examine two approaches, the *Thread Level Parallelism* (TLP) approach using OpenMP and the *Single Instruction Multiple Data* (SIMD) approach using Intel SSE Intrinsics. Using an Intel Core 2 series workstation with 4 cores and a SIMD vector width of 128 bits (4 integers) we optimized the following simple loop, which is amenable to both the TLP and SIMD approaches:

```
void inc_all(int array[], int n) {
  for(int i = 0; i < n; ++i)
    array[i] = array[i] + 1;
}
```

Table 1 compares the performance and power tradeoffs. The first consideration is the maximum speedup obtained under ideal circumstances (the same for both approaches, $4\times$). In practice, for large data sets (32K elements) the thread based approach slightly outperforms the SIMD approach. However, when the input data size is small (between 8-64 elements) the thread based approach actually results in a slowdown of $26\times$, due to thread management overhead, while the SIMD based approach is still able to achieve a speedup of $2.4\times$. In fact the thread based approach does not break even in runtime until the data size is over 2000 elements while the SIMD approach shows consistent performance improvements starting with a data size of 4 elements. An additional concern when considering these approaches is that the TLP approach must power up multiple cores, greatly increasing the power consumption, while the SIMD approach utilizes a small amount of extra hardware in the execution units of a core.

Feature	TLP	SIMD
Max Speedup	$4\times$	$4\times$
Speedup (32K)	$3.1\times$	$2.9\times$
Speedup (8-64)	$26\times$ slower!	$2.4\times$
Break Even Size	2000	4
Power Impact	$4\times$ core power	negligible
Generality	High	Limited

Table 1: Attributes of TLP and SIMD approaches

Despite these advantages SIMD based approaches have historically been seen as less generally applicable than TLP based approaches. The reason: many programs are *irregular*—they contain heavy branching, hard to predict loop exit conditions, and sparse data layouts—all of which complicate the use of SIMD operations. Irregularity has historically limited the application of SIMD operations to high performance computing or specialized algorithms (e.g. video processing). We claim irregularity is not insurmountable and that by enabling the application of SIMD operations in more contexts we can obtain large increases in performance with negligible increases in power usage, even for irregular code.

SIMD with Irregular Control Flow. A standard approach to dealing with control flow irregularity is to speculate past control dependencies and only use the results we need from that speculation. In order to determine if this approach can be effectively applied using SIMD operations we looked at a selection of algorithms from the C++ *Standard Template Library (STL)*. We focused primarily on algorithms which have non-trivial control flow and abnormal exit conditions (i.e., `find`, `reverse`, `equals`, etc.).

For small input sizes (4 - 32 elements) the observed speedup was between 5% and 20%, while for larger input sets (512+ elements) the speedup approached a factor of 2-3$\times$. Further, by simply compiling the 483.xalan benchmark (from SPEC CPU) with the SIMD version of the STL, we saw a 5% reduction in total runtime. Leveraging recent advances in *SAT Modulo Theory* (SMT) provers and work on *Relational Program Verification* we were able to verify the semantic equivalence of the reference implementations and our SIMDized versions. Thus, demonstrating that not only could core data structure libraries be effectively optimized via SIMD operations (and that these optimizations have a non-trivial impact in real code) but also that automated techniques could be used to to precisely reason about the code.

Another recent development in the application of SIMD operations to irregular computations is the introduction of additional hardware support. In particular the SSE 4.2 instruction set contains a number of operations specifically for comparing and searching in strings which move much of the data speculation logic into the hardware. Thus, it becomes trivial to implement vectorized string operations (such as `find_first_of(char c)` or even `find_first_of(string& str)`) that outperform a baseline implementation by factors of $8\times$ or more.

SIMD with Data Restructuring. In order to fully realize the performance potential of SIMD operations, we must also address challenges related to the layout of data. In particular, SIMD requires that data be contiguously laid out in memory. Consider a binary search algorithm and how we exploit data parallelism in it:

```
int binary_search(int array[], int n, int key) {
  int min = 0, max = n;
  do {
    int middle = (min + max) / 2;
    if (key > array[middle]) min = middle + 1;
    else max = middle;
  } while(min < max);
  return array[min];
}
```

Let's suppose a programmer calls binary_search with n=64. On the first iteration of the loop middle is 32. while on the second iteration middle is *either* 16 or 48. While we may not know the exact value middle will take on the second iteration of the loop, we know it will be either 16 or 48. Thus, we can execute the first two iterations of the loop by using a single SIMD instruction to compare key against the three elements in array at indexes 32, 16, and 48, respectively. We then extract from the result of this SIMD instruction the values we need in order to move on to the third iteration of the loop. For example, if key happens to be less than the value at array[32] we ignore the comparison against the value at array[48] and update min and max based upon the result of the comparison of key against the value at array[16].

While this addresses the control flow dependency issues, we end up loading array elements that are scattered throughout memory. This prevents the use of the parallel load SIMD operation and serializes the algorithm's execution. To address this, while sorting the array—a precondition of binary search—we co-locate elements of the array that are accessed in successive iterations of the algorithm. This contiguous layout allows parallel loading of the data elements needed for the each of the comparisons. In our current implementation we execute 4 iterations of the loop at a time with SIMD instructions and obtain a $20\times$ speedup of binary search.

Conclusions. There is a rich and successful history of applying SIMD operations to regular computations. Work on compilation of these types of applications provides a solid basis of techniques for loop transformation, control flow dependency elimination, and alignment optimizations [3, 5, 7, 6] that are needed for effective vectorization in a compiler [2, 4]. This work, combined with the information in developer guides for manual vectorization [1], shows that when vectorization can be applied the performance gains are substantial.

Given our initial success we believe there is great potential for further research in techniques for applying SIMD operations to code that may initially appear too irregular to contain data parallelism. These techniques could include the development of programming language constructs to support vectorization, tools for automated reasoning about vectorized code, analysis techniques for restructuring data layouts for vectorization, and investigation into additional hardware support, among other research. We believe that there is a confluence of events, a plateauing of clock speed driven performance improvements, the widespread availability of rich SIMD instruction sets, and growing concerns about power consumption, which makes increasing single-core performance a fruitful area of investigation, even in the age of multi-core.

References

[1] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/products/processor/manuals/, April 2011.

[2] K. Kennedy and J. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2002.

[3] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[4] D. Nuzman and R. Henderson. Multi-platform auto-vectorization. In *CGO*, 2006.

[5] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *PACT*, 2008.

[6] J. Shin, M. Hall, and J. Cha. Superword-level parallelism in the presence of control flow. In *CGO*, 2005.

[7] P. Wu, A. Eichenberger, and A. Wang. Efficient SIMD code generation for runtime alignment and length conversion. In *CGO*, 2005.