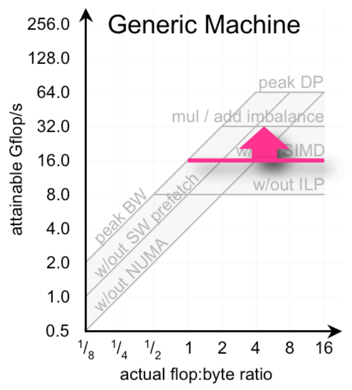
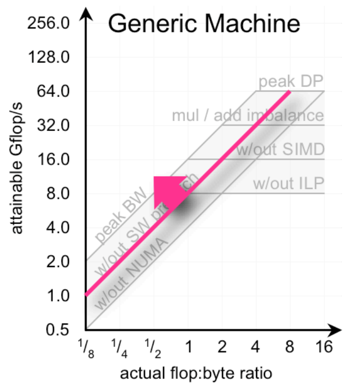
Appendix A

Given the short page limit of papers for CACM, we've included this appendix to provide the details of the Roofline methodology, as well as discussions on subtle secondary issues and our future work directions.

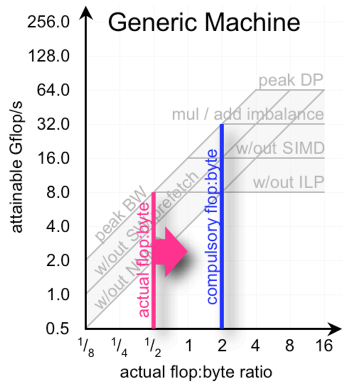
To put the model into a global perspective, Figure A1 shows the general goals of program optimization: increasing computational performance, increasing memory performance, and increasing operational intensity. When left of the ridge point, increasing operational intensity improves performance by increasing locality.



(a) maximizing in-core performance



(b) maximizing bandwidth



(c) minimizing traffic

Figure A1. General ways to improve performance in the Roofline model.

A.1 Finding Operational Intensity, Rooflines, and Ceilings

A DRAM bandwidth-oriented Roofline model is built using three sets of numbers collected either from microbenchmarks or derived from a given architecture's software optimization manual. In general, performance is the minimum of:

1. Op. Intensity * Bandwidth (with optimizations 1…i)
2. In-core Flop/sec (with optimizations 1…j)
3. In-core Flop/sec as a function of the floating-point fraction.

Typically one of the last two dominates on a given architecture. As such, we draw only one Roofline per machine. These parameters provide kernel-independent bounds to performance. Thus, these parameters are collected independently only once per machine per metric. This section details how these ceilings are either measured or calculated.

A.1.1 Operational Intensity

True operational intensity is both architecture- and kernel-dependent and thus must be calculated for every kernel-architecture combination. Perhaps the easiest way to calculate operational intensity is to use performance counters to measure the actual number of operations and to measure the actual amount of memory traffic when running the kernel. In practice, depending on the kernel, it may be easy to calculate both the number of interesting operations and the minimum memory traffic by hand. Thus, one can bound the operational intensity.

A.1.2 Main Memory Bandwidth

The first set of ceilings is main memory bandwidth with increasing optimization. Although the STREAM benchmark claims to report this bandwidth, it does not. It actually measures performance in terms of iterations per second, and then attempts to convert this to bandwidth based on the compulsory memory traffic on a non-write allocate architecture. This subtle, yet critical difference implies it cannot account for either conflict misses or the traffic associated with a fill on a write miss.

To correctly measure streaming bandwidth, we wrote a series of highly tuned versions of the STREAM benchmark that perform both a dot product and a copy. We pad arrays to avoid both bank and cache conflicts. We exploit the cache bypass instructions or increase the conversion constant to account for the fill traffic. The most naïve implementation allocates all data on one processor (no memory affinity), but is appropriately unrolled and padded. We proceed by correctly exploiting memory affinity and collect a new bandwidth. We then add software prefetching with an auto-tuned prefetch distance to the loop and measure bandwidth. Finally, we attempt to reduce the data set size to improve the effectiveness of a snoop filter. This provides a fourth bandwidth. We benchmark these individually, and define a new ceiling for each measured bandwidth.

A.1.3 In-Core Parallelism

To estimate performance as a function of exploited in-core parallelism we rely on the appropriate software optimization manual for the architecture in question. In the long term, this is not a productive solution, as one would need to be very familiar with the breadth and evolution of all current and future architectures. However, for the purposes of this paper, no benchmark was necessary.

Consider the following reduction:

$$y = x[1] + x[2] + x[3] + \ldots + x[N]$$

We define thread-level-parallelism as the simplest parallelization optimization that could be applied. As such, the lowest ceiling is defined as the thread-level-parallelism-only ceiling. Each thread receives *N*/*NThreads* elements. We assume each thread executes a naïvely unrolled, yet dependent chain of scalar floating-point adds. Thus there is no instruction-, data-, or functional unit-level parallelism in the lowest ceiling. As such, the next add in the chain cannot be started until the previous has been completed. As a result, the latency of the floating-point pipeline is exposed. The resultant bound on throughput, irrespective of bandwidth, is calculated as:

$$\mathit{Cores} \times \mathit{Frequency} \times \mathit{max}(1, \mathit{ThreadsPerCore}/\mathit{Latency})$$

Where *ThreadsPerCore* is the number of cores sharing a FPU within a core on a fine-grained multithreaded architecture. With enough threads, the FPU can be full utilized with *Latency* threads hiding the FPU latency.

If the loop were further optimized by unrolling and maintaining several partial sums, then instruction-level parallelism is expressed. Thus, the next ceiling assumes sufficient per thread instruction-level parallelism to hide the functional unit latency. SIMD may not be included. Thus, the FPU would be completely occupied with scalar adds. The resultant throughput is:

$$\mathit{Cores} \times \mathit{Frequency}$$

Third, we add data-level parallelism (SIMD) to the mix. Thus every two scalar add instructions into the partial sums becomes one SIMD add instruction in which two partial sums are stored in a SIMD register. For arbitrary SIMD register width, the resultant ceiling that incorporates thread-, instruction, and data-level parallelism is calculated as:

$$\mathit{Cores} \times \mathit{Frequency} \times \mathit{SIMD\ width} \,/\, \mathit{SIMD\ throughput}$$

The throughput term must be included as some architectures support SIMD instructions, but execute only one element per cycle. Thus, for an older Santa Rosa Opteron processor executing double precision SIMD instructions, the width is 2 flops per instruction and throughput is one instruction per two cycles.

Notice, the code does not perform any floating-point multiplies. However, if it were changed to:

$$z = y[1]*x[1] + y[2]*x[2] + y[3]*x[3] + \ldots + y[N]*x[N]$$

Then there would be essentially a balance between the number of multiplies and adds. As such, we define peak in-core performance as the execution of unrolled and SIMDized fused multiply adds (FMAs); that is, the simultaneous execution of multiplies and adds. For architectures with FMA or parallel add and multiply datapaths, the resultant bound on in-core performance is:

$$2 \times \mathit{Cores} \times \mathit{Freq.} \times \mathit{SIMD\ width} \,/\, \mathit{SIMD\ throughput}$$

On Niagara2, on the other hand, each core may issue only one scalar floating-point instruction per cycle, so the bound is simply:

$$\mathit{Cores} \times \mathit{Frequency}$$

Note that some computers, such as the IBM P5, have multiple, identical floating point datapaths; ILP would be used to satisfy both superscalar and deeply pipelined functional units. As such, they could get even more benefit from greater ILP than these equations show.

A.1.4 Instruction Mix

All processors have limited instruction issue bandwidth. Their floating-point issue bandwidth is less than or equal to this bandwidth. As the non-floating-point fraction of issued instructions increases, eventually floating-point issue bandwidth will be starved to serve non-floating-point instructions. We calculate an orthogonal set of ceilings based on the floating-point fraction of the instruction mix assuming full exploitation of in-core parallelism. This approach is somewhat complicated as on Cell a double precision instruction stalls the issue unit for a further 6 cycles. We delineate the floating-point fraction in negative powers of 2. For a given architecture and kernel it is usually clear which in-core ceilings should be used. Such ceilings account for the potentially limited integer performance of these machines.

A.2 LOAD BALANCE AND ROOFLINE

Load balance can loosely be categorized as either imbalance in the memory accesses or imbalance in the computation.

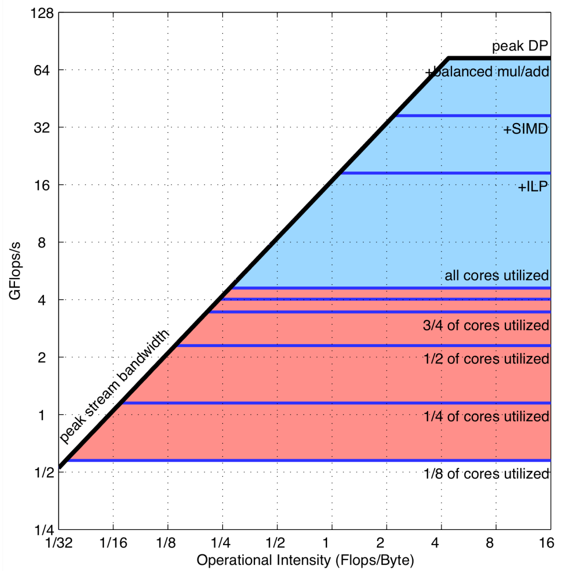


Figure A2. Roofline for AMD Opteron X4 where first step is to load balance, then to optimize.

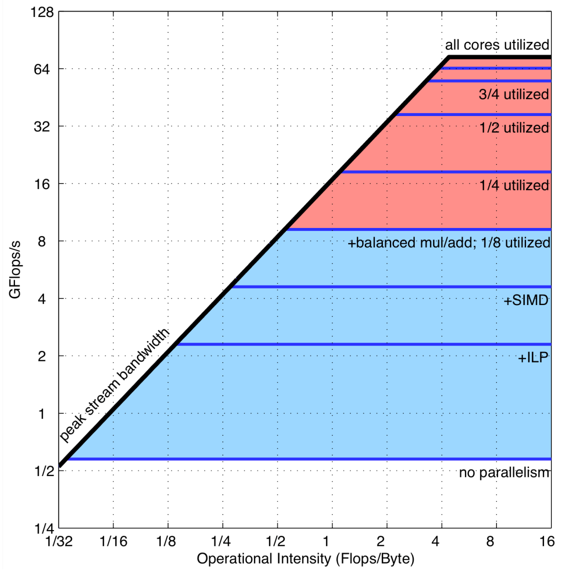


Figure A3. Roofline for AMD Opteron X4 where first step is to optimize performance within a core, and then to load balance.

A.2.1 Computation Imbalance

Computational imbalance is easily visualized and understood. As imbalance increases, fewer and fewer threads must do all the work. In the limiting case, exempting poor barrier implementations, performance is sequential. Thus, we may define log2P ceilings denoting powers-of-two load imbalance bounds on performance. Depending on whether load balancing is perceived as a more tractable problem than in-core optimization, it can be placed either directly below the roofline or below the TLP only ceiling. Figures A2 and A3 show the two approaches to a Roofline model for load balancing computation.

A.2.2 Memory Imbalance

Memory imbalance occurs when the main memory traffic generated by one core is dramatically different than another or when some of the memory controllers are much more heavily loaded than others. Previously, we explored the latter in the case of memory affinity. When all of the data is located with one socket of a shared memory multiprocessor, there is a clear imbalance in the load on the memory controllers—the controllers of the other sockets are unused. In the context of the Roofline model, a bandwidth ceiling denotes this diminished performance.

In the context of imbalance in the memory traffic generated per core, Little’s Law is not being satisfied. The same concurrency is required based on the latency-bandwidth product. However, cores that do not generate any main memory traffic diminish the chip-wide concurrency that can be exploited. When the exploited concurrency dips below the requisite concurrency to satisfy Little’s Law, sustained bandwidth decreases. For our SPMD codes, this imbalance never happened.

However, in general one could visualize this as a series of progressively lower bandwidth ceilings labeled by the fraction of cores generating main memory traffic. These could either be placed below the roofline or below the lowest ceiling depending on which is perceived as easier to achieve: memory optimizations or load balance. A benchmark is required to generate such a figure. Figure A4 shows memory imbalance.

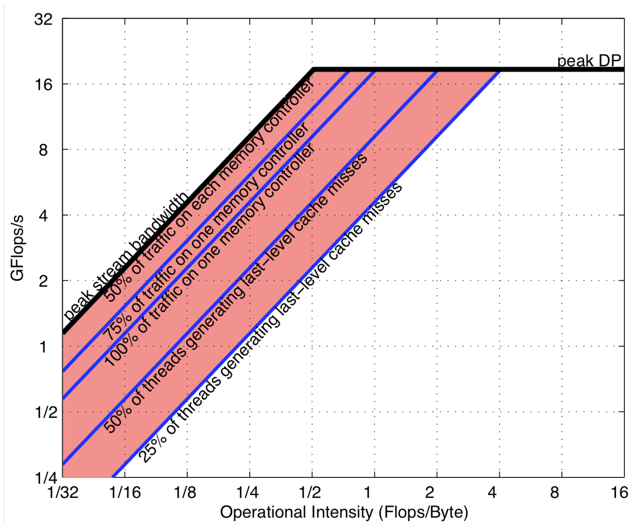


Figure A4. Roofline for Sun UltraSPARC T2 where memory accesses are unbalanced.

A.3 INTERACTION WITH PERFORMANCE COUNTERS

The Roofline as drawn shows the benefit of full exploitation of each architectural paradigm. The good news is that this model gives insight to the architect, compiler writer, and programmer as to what are the strengths and weaknesses of a system.

If one gets 100% of ILP, 100% if DLP, and 50% of multiply/add balance it is easy to estimate performance. However, in practice it might not be possible to fully exploit all but one feature. In reality, one might exploit 85% of ILP, 75% of SIMD, and have 65% balance between multiplies and adds.

Hence, an interesting future direction is to supplement the “architecture-oriented” Roofline model presented above is to use performance counters to generate a “runtime-oriented” Roofline model. One could start from the base Roofline and use performance counters to generate ceilings that represent how much performance was lost due to not exploiting the various architectural features. For example, one could examine the performance counter that counts how many floating-point SIMD instructions were issued. Dividing this by the total number of issued floating-point instructions would define a true SIMD ceiling. To be clear, if no SIMD instructions were issued, then the ceiling would equal half the peak performance, but if all instructions issued were SIMD, then the ceiling would be the peak. It is critical that when calculating in-core ceilings, stalls from memory be ignored.

Performance counters could also be used to estimate the true limitations to peak bandwidth. It is easy to calculate bandwidth by counting the total DRAM memory traffic across all memory controllers. By using performance counters to note imbalance among memory controllers, one could estimate the benefit of further memory affinity optimizations. Similarly, one could count the latency cycles when queues aren't full to determine the actual potential of software prefetching.

Finally, performance counters could be used to determine the true operational intensity. Ideally, performance counters could distinguish compulsory misses from capacity or conflict misses. As such one could decide if cache optimizations are likely to be beneficial. Moreover, if one could distinguish capacity misses from conflict misses, one could decide whether cache blocking or array padding optimizations are likely to show benefits.

Figure A5 shows the traditional architectural-oriented model of the Opteron X4, while Figure A6 shows the runtime-oriented Roofline model for the Opteron X4 for a hypothetical kernel.

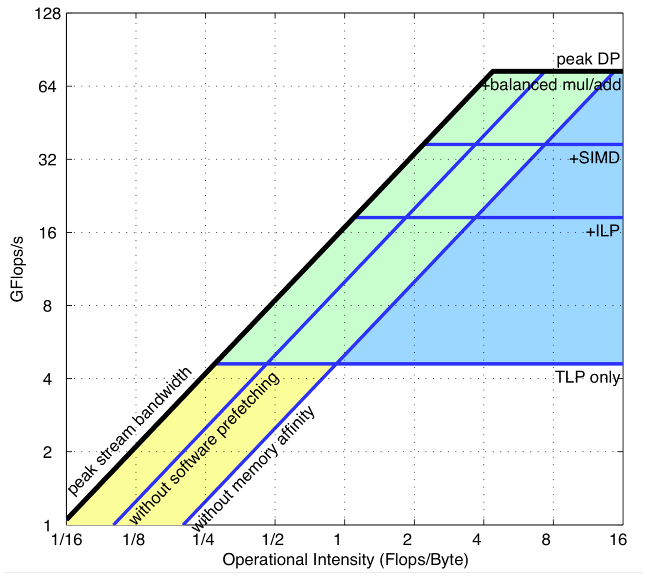


Figure A5. Traditional Architecture-Oriented Roofline Model for the Opteron X4, as presented earlier in the paper.

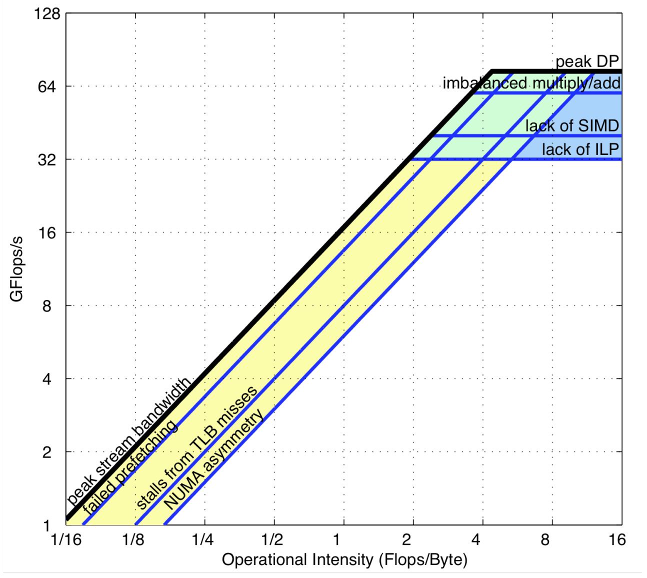


Figure A6. Runtime-Oriented Roofline Model of the Opteron X4, in contrast Architecture-Oriented Model in Figure A5.