Many-Core vs. Many-Thread Machines: Stay Away From the Valley

Zvika Guz[1], Evgeny Bolotin[2], Idit Keidar[1], Avinoam Kolodny[1], Avi Mendelson[3], and Uri C. Weiser[1]
[1]*EE Department, Technion, Israel* [2]*Intel Corporation* [3]*Microsoft Corporation*
[1]{zguz@tx, idish@ee, kolodny@ee, uri.weiser@ee}.technion.ac.il, [2]evgeny.bolotin@intel.com, [3]avim@microsoft.com

Abstract—We study the tradeoffs between Many-Core machines like Intel's Larrabee and Many-Thread machines like Nvidia and AMD GPGPUs. We define a unified model describing a superposition of the two architectures, and use it to identify operation zones for which each machine is more suitable. Moreover, we identify an intermediate zone in which both machines deliver inferior performance. We study the shape of this "performance valley" and provide insights on how it can be avoided.

Index Terms—Chip Multiprocessors, GPGPU

1. Introduction

As chip multiprocessors are rapidly taking over the computing world, we see the evolution of such chips progressing along two separate paths. At one end of the spectrum, general purpose uni-processors have evolved into dual- and quad-cores, and are set to continue this trajectory to dozens of cores on-chip. Such machines follow the legacy of single cores in using caches to mask the latency of memory access and reduce out of die bandwidth, and typically dedicate a significant portion of the chip to caches. We call these *Many-Core (MC) machines*. Intel's Larrabee [8] is a prominent example of this approach. At the same time, processor engines that can run numerous simple threads concurrently, which were traditionally used for graphics and media applications, are now evolving to allow general-purpose usage [7]. The latter usually do not employ caches, and instead use thread level parallelism to mask memory latency, by running other threads when some are stalled, waiting for memory. We refer to these as *Many-Thread (MT) machines*. Examples of such machines are the current GPGPUs of Nvidia and AMD.

To date, the tradeoffs (and even the boundaries) between these approaches are not well formalized. Such formalization and understandings are, nevertheless, essential for processor architects, who need insights on how to improve their machine's performance on a broad range of workloads. In this paper, we take a step towards understanding the tradeoffs between MC and MT machines, and the domains where each is more appropriate.

To this end, we define (in Section 2) a simple unified model of the two architectures. The model captures an imaginary hybrid machine, comprised of many (e.g., 1024) simple (in-order) processing elements (PEs) and a large (e.g., 16MB) shared cache. The model considers a number of parameters, such as number of PEs, cache size, cache and memory latencies, etc. We then provide an equation for deducing the performance for each set of parameters.

When instantiated with a modest number of independent threads (say, up to a few hundreds), the model approximates MC machines, where the cache is large enough to cater to all threads. With a very large number of independent threads (in the thousands), the same model more closely describes an MT machine, since the cache is no longer effective, and the memory access latency is masked by the increased thread-level parallelism.

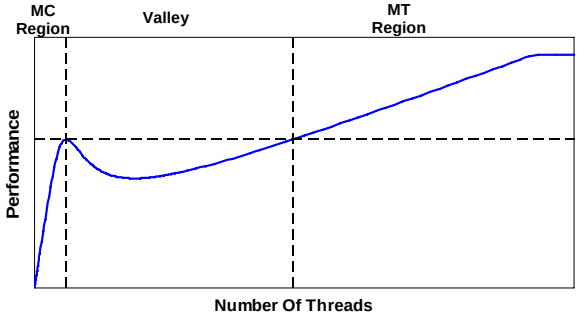

Fig. 1. Performance of a unified many-core (MC) many-thread (MT) machine exhibits three performance regions, depending on the number of threads in the workload.

Our results not only show these two distinct performance regions, but also show that there is a *valley* between them, where performance is worse than at both regions. Fig. 1 illustrates this phenomenon. We see that in the (leftmost) *MC region*, as long as the cache capacity can effectively serve the growing number of threads, increasing the number of threads improves performance, as more PEs are utilized. At some point the cache becomes too small for covering the growing stream of access requests. Memory latency is no longer masked by the cache, and performance takes a dip into the valley. The valley represents an operation point (number of threads) where both MC and MT perform poorly, as neither can mask the memory access latency. However, as the number of threads increases, the *MT region* is reached, where the thread coverage is high enough to mask the memory latency. In this region, (in an unlimited memory bandwidth environment), performance continues to improve, up to the maximal performance of the machine.

The question of how performance depends on the de-

gree of multithreading was studied in the early 90's by Agrawal [1]. In retrospect, Agrawal's analysis can be seen as applicable to our MC region, and it observes a similar trend to the one exhibited in the leftmost area of our curve: with the first few threads, performance soars, but then hits a plateau, as caches become less effective. Given (dated) parameter values from the 90's, Agrawal found that as little as two or four threads are sufficient to achieve high processor utilization. Our work takes the level of parallelism much further, to tens of thousands of threads, and observes that the plateau is followed by a valley, and then by another uphill slope (the MT region), which in some cases even exceeds the MC peak.

While the exact shape of the curve depends on numerous parameters, the general phenomenon is almost universal. This illustrates the major challenge that processor designers today face - how to stay away from the valley? Indeed, the challenge for MC designers is to extend the MC area to the right and up, so as to be able to exploit higher levels of parallelism. The challenge for their MT counterparts is to extend the MT zone to the left, so as to be effective even when a high number of independent threads is not available. In Section 3, we discuss how various parameters (of the machine or the workload) impact the shape of the valley, providing insights on how the above challenges may be addressed.

Finally, we note that we focus on workloads that can be parallelized into a large number of independent threads with practically no serial code. It is already well-understood that for serial code, MC machines significantly outperform MT ones, and that for applications that alternate between parallel and serial phases asymmetric machines are favorable [4][6]. Our model instead focuses on workloads that offer a high level of parallelism, where the questions we set to answer are still open.

2. MT/MC UNIFIED MODEL

In order to study the Many-Cores/Many-Threads tradeoff, we present a unified model describing a superposition of the two architectures. We provide a formula to predict performance (in *Giga Operations Per Second-GOPS*), given the degree of multithreading and other factors. Our model machine is described by the following parameters:

- N_{PE} - *Number of PEs. (Simple, in-order processing elements.)*
- $S_{\$}$ - *Cache size [Bytes]*
- CPI_{exe} - *Average number of cycles required to execute an instruction assuming a perfect (zero-latency) memory system. [cycles]*
- t_m - *Memory latency [cycles]*
- $t_{\$}$ - *Cache latency [cycles]*
- f - *Processor frequency [Hz]*
- r_m - *Ratio of memory instructions out of the total instruction mix ($0 \le r_m \le 1$).*

Given the above notations, we get that once every $1/r_m$ instructions, a thread needs to stall until the data it accesses is received from memory. We denote:

- t_{avg} - *Average time needed for data access [cycles]*

(t_{avg} is developed to account for the probability of finding the data in the cache in Equation (3).)

During this stall time, the PE is left unutilized, unless other threads are available to switch-in. The number of additional threads needed in order to fill the PE's stall time is $\frac{t_{avg}}{CPI_{exe}/r_m}$, and hence $N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)$ threads are needed to fully utilize the entire machine.

Processor utilization ($0 \le \eta \le 1$) (i.e., the average utilization of all PEs in the machine) is given by:

$$\eta = \min\left(1 \ , \ \frac{n_{threads}}{N_{PE} \cdot \left(1 + t_{avg} \cdot \frac{r_m}{CPI_{exe}}\right)}\right) \qquad (1)$$

where $n_{threads}$ is the degree of multithreading (the number of threads available in the workload). Our model assumes that all threads are independent of each other (as explained in Section 1), and that thread's context is saved in the cache (or in other dedicated on-chip storage) when threads swap out. The machine can thus support any number of in-flight threads as long as it has enough storage capacity to save their contexts. For simplicity, Equation (1) neglects the thread swap time, though it can be easily factored in. The minimum in Equation (1) captures the fact that after all execution units are saturated, there is no gain in adding more threads to the pool.

Finally, the expected performance is:

$$Performance\,[GOPS] = N_{PE} \cdot \frac{f}{CPI_{exe}} \cdot \eta \qquad (2)$$

The system utilization (η) is a function of two variables, $n_{threads}$ and t_{avg}, where t_{avg} is affected by the memory hierarchy (the access times of caches and external memory) and the behavior of the workload (which determines cache hit rate). t_{avg} can be approximated as follows:

$$t_{avg} = P_{hit}(S_{\$}^{thread}) \cdot t_{\$} + (1 - P_{hit}(S_{\$}^{thread})) \cdot t_m \ , \qquad (3)$$

where $P_{hit}(S_{\$}^{thread})$ is the hit rate ratio for each thread in the application assuming it can exploit a cache of size $S_{\$}^{thread}$. Given a shared pool of on-chip cache resources, the cache size available for each thread decreases as the number of threads grows. Hence, Equation (3) can be rewritten as:

$$t_{avg} = P_{hit}\left(\frac{S_{\$}}{n_{threads}}\right) \cdot t_{\$} + \left(1 - P_{hit}\left(\frac{S_{\$}}{n_{threads}}\right)\right) \cdot t_m \qquad (4)$$

Equation (4) assumes that threads do not share data – we leave the effect of sharing for future work. It also ignores the fact that threads contexts, saved in the cache, reduce the overall storage capacity left to hold data. This effect proved to be negligible in our example and is hence neglected here for the sake of simplicity. Notice, though, that it can be easily factored in.

We deliberately refrain from presenting a concrete P_{hit} function. Any hit rate function (derived either from simulations or from an analytical model) may be used here without undermining the rest of the discussion. In the next section, we present the results for several specific hit rate functions.

3. RESHAPING THE VALLEY

Both MC machines and MT machines strive to stay away from the performance valley, or to flatten it. In this section, we study how different parameters affect the performance plot, via an example design. We first present an example system and an analytical hit rate function chosen to characterize the workload. We then exemplify how workload attributes such as compute intensity (Section 3.1); and how hardware attributes such as memory latency (Section 3.2) and cache size (Section 3.3), effect the shape of the performance plot. While in these sections we assume for the sake of clarity an unlimited bandwidth to memory, we consider bandwidth to be a principal performance limiter and account for the effect of a limited off-chip bandwidth in Section 3.4.

The example system we use in this section consists of 1024 PEs and a 16MB cache. We assume a frequency of 1GHz, a CPI_{exe} of 1 cycle, and an off-chip memory latency (t_m) of 200 cycles. The peak performance of the example machine is 1 Terra OPS (Equation (2) with η =1). In our baseline workload, 1 out of 5 instructions is a memory instruction, i.e. r_m=0.2.

The problem of finding an analytical model for the cache hit rate function has been widely studied ([1][2][9] among others). In this paper, we use the following simple function [5]:

$$P_{hit}(S_\$) = 1 - \left(\frac{S_\$}{\beta} + 1\right)^{-(\alpha-1)} \qquad (5)$$

The function is based upon the well known empirical power law from the 70's (also known as the 30% rule or √2 rule) [3]. In Equation (5), workload locality increases when increasing α or decreasing β.

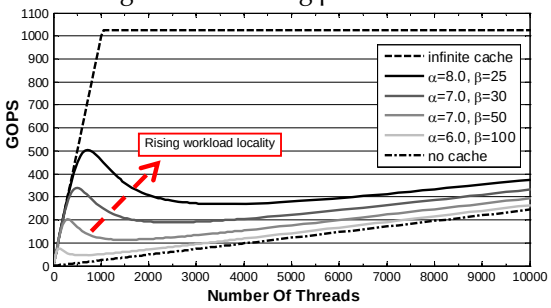

Fig. 2. Performance for different cache hit rate functions.

Fig. 2 presents the projected performance in GOPS as a function of number of threads, for different values of α and β. It also presents the upper and lower limits – a perfect cache (dashed line, all accesses are satisfied from the cache) and no cache at all (dotted line, all accesses are satisfied from main memory). For the rest of the paper, we use α=7 and β=50.

3.1 Compute/memory Ratio Impact

We explore how the compute intensity of the workload (measured in the ratio between compute instructions and memory accesses, i.e., $(1-r_m)/r_m$) affects the shape of the performance plot. Fig. 3 presents performance for different compute/memory ratios.

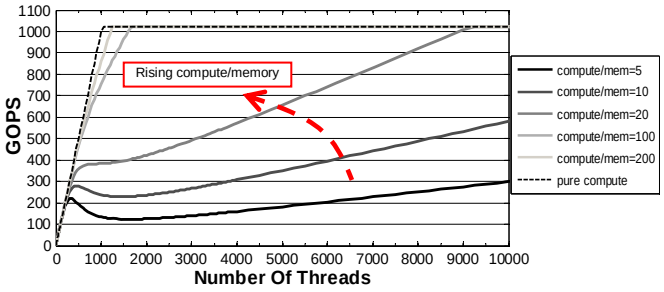

Fig. 3. Performance for different compute/memory ratios.

Fig. 3 shows that the more computation instructions per memory instructions are given, the steeper performance climbs. The above trend results from the fact that as more instructions are available for each memory access, fewer threads are needed in order to fill the stall time resulting from waiting for memory. Moreover, the penalty of accessing memory is amortized by the small portion of accesses out of the total instruction mix. All in all, a high compute/memory ratio decreases the need for caches, and eliminates the valley.

Applications with more computation per memory access (for example the light-gray lines) reach peak performance much faster, and can even avoid the valley entirely since there is no significant memory latency to screen. Moreover, in such applications there is practically no difference between MC and MT as caches have almost no effect. In other words, a higher value of the compute/memory ratio makes the workload more suitable for MT machines. This trend has been identified by MT machine makers, who suggest aiming for a high compute/memory ratio as a programming guideline [7].

3.2 Memory Latency Impact

We next examine how the latency to off-chip memory (given in number of cycles needed to access the data, i.e. t_m) affects the shape of the performance plot. Fig. 4 presents the performance plot for several different off-chip memory latencies.

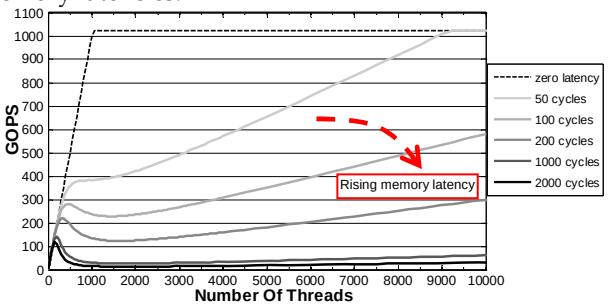

Fig. 4. Performance for different off-chip latencies.

Memory latency is important in the MT region, as observed in the gradient of performance improvement. Memory latency is also important in the MC region since a shorter latency draws the MC peak higher. With the scaling of process technology, the memory latency gap is expected to grow, thus pushing the plots down.

3.3 Cache Size Impact

Next, we examine the effect of cache size (i.e., $S_\$$) on the

performance graph. Fig. 5 presents several performance plots differing in their cache size.

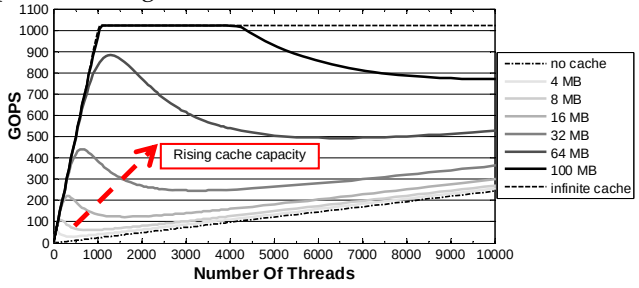

Fig. 5. Performance for different cache sizes.

As can be seen in Fig. 5, cache size is a crucial factor for MC machines, as larger caches extend the MC area up and to the right. This is because larger caches suffice for more in-flight threads, thus enabling the curve to stay longer with the "perfect cache" scenario.

3.4 Off-Chip Bandwidth Impact

In the previous sections, we assumed an unlimited bandwidth to external memories. Alas, off-chip bandwidth is a principal bottleneck and may limit performance as we show in this Section. In order to take into account the role of off-chip bandwidth, we present the following new notations:

b_{reg}- *Size of operands [Bytes]*

BW_{max}- *Maximal off-chip bandwidth [GB/sec]*

The latter is an attribute of the on-die physical channel. Using the above notations, off-chip bandwidth can be expressed as:

$$BW = Performance \cdot r_m \cdot b_{reg} \cdot (1 - P_{hit}) \quad , \qquad (6)$$

where performance is given in GOPS.

Given a concrete chip with a peak off-chip bandwidth limit (BW_{max}), the maximal performance achievable by a machine is $BW_{\max} / \left(r_m \cdot b_{reg} \cdot (1 - P_{hit}) \right)$. Fig. 6 presents the performance plot with several different bandwidths limits, assuming all operands are 4 bytes long. (i.e., single-precision calculations.)

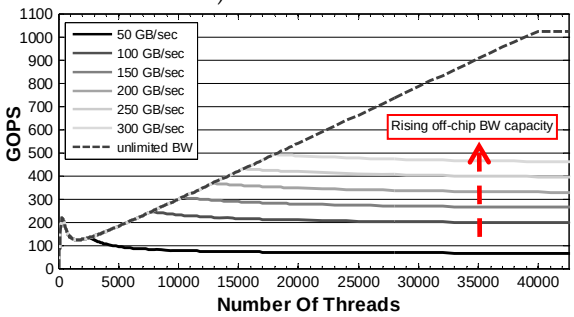

Fig. 6. Performance with limited off-chip bandwidth.

At the rightmost side of the plot, where all accesses to data are served from memory, performance converges to $BW_{\max} / (r_m \cdot b_{reg})$. However, some of the plots exhibit reduction in performance after reaching a peak at the MT region. To explain why this happen recall that P_{hit} is affected by the number of threads in the system, because the more in-flight threads there are, the less cache is available to each one of them. Therefore, when the off-chip bandwidth wall is met, adding more threads only degrades performance due to increasing off-chip pressure.

The bandwidth wall in our example causes some of the plots to never reach the point where the MT area exceeds the performance of the MC area. This is because the BW wall limits the performance of the MT region but not that of the MC region, where caches can dramatically reduce the pressure on off-chip memories. In other words, MT machines, relying on very high thread counts to be effective, dramatically increase the off-chip bandwidth pressure, and are hence in need of very aggressive memory channels to be able to deliver performance (e.g., up to 150GB/sec in Nvidia's current GPUs). MC machines, on the other hand can suffice with relatively slimmer channels, as their caches screen out most of the data accesses (e.g., 20-30GB/sec in Intel's Nehalem processor).

4. Summary

We studied the performance of a hybrid Many-Core (MC) and Many-Thread (MT) machine as a function of the number of concurrent threads available. We found that while both MC and MT machines may shine when given a suitable workload (in number of threads), both suffer from a "performance valley", where they perform poorly compared to their achievable peaks. We studied how several key characteristics of both the workload and the hardware impact performance, and presented insights on how processor designers can stay away from the valley.

Acknowledgment

We thank Ronny Ronen, Michael Behar, and Roni Rosner. This work was partially supported by Semiconductors Research Corporation (SRC), Intel, and the Israeli Ministry of Science Knowledge Center on Chip MultiProcessors.

References

[1] A. Agrawal, "Performance Tradeoffs in Multithreaded Processors," IEEE Trans. on Parallel and Distributed Systems, 1992

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," ACM Trans. on Computer Systems, May 1989

[3] C. K. Chow, "Determination of Cache's Capacity and its Matching Storage Hierarchy," IEEE Trans. Computers, vol. c-25, 1976

[4] M. D. Hill, and M. R. Marty, "Amdahl's Law in the Multicore Era," IEEE Computer, vol. 46, July 2008

[5] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An Analytical Model for Designing Memory Hierarchies," IEEE Trans. Computers, vol. 45, no 10, October 1996

[6] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguadé, "Performance, Power Efficiency, and Scalability of Asymmetric Cluster Chip Multiprocessors," Computer Architecture Letters, vol. 4, July 2005

[7] NVIDIA, "CUDA Programming Guide 2.0," June 2008

[8] L. Seiler, et al., "Larrabee: a many-core x86 architecture for visual computing," SIGGRAPH 2008

[9] D. Thiebaut, and H. S. Stone, "Footprints in the cache," ACM Trans. on Computer Systems (TOCS), Nov. 1987