SCALABILITY OF MULTIMEDIA APPLICATIONS ON NEXT-GENERATION PROCESSORS

Guy Amit[1], Yaron Caspi[1,2], Ran Vitale[1] and Adi T. Pinhas[1]
[1]Corporate Technology Group, Intel Corp, Israel [2]Weizmann Institute of Science, Israel
{guy.amit, adi.pinhas}@intel.com

ABSTRACT

In the near future, the majority of personal computers are expected to have several processing units. This is referred to as Core Multiprocessing (CMP). Furthermore, each of the computation units will be capable of running multiple hardware threads. To benefit from the additional processing power, application developers should multithread their software. This paper studies the scalability (expected speedup factor) of multimedia applications and provides guidelines for proper utilization of these new multi-core platforms. In particular, we discuss the decomposition method, load balancing, synchronization primitives, interaction with the operating system and hardware issues such as cache hierarchy and memory bandwidth. Our results are based on analysis of several state-of-the-art applications, including H.264 video encoding, panoramic image stitching and dense optical-flow estimation. We demonstrate how to multithread them properly, and report scalability results on several next-generation multi-core platforms.

1. INTRODUCTION

Over the past several years, a major factor in improving processor performance has been micro-architecture mechanisms that exploit different levels of parallelism in the program. One approach, named *Instruction-Level Parallelism* (ILP), is to execute multiple instructions concurrently on several execution units. Another approach ,termed *Data-Level Parallelism* (DLP), is to execute a specific operation on multiple data elements within a single instruction (e.g. MMX). Recent processors support *Thread Level Parallelism* (TLP). These processors are able to run two or more threads simultaneously. Future processors are expected to have a larger number of cores on die.

This paper studies the expected speedup that these architectures can provide for high-performance applications. Multimedia workloads typically have independent kernels and steady computation patterns that enable functional decomposition. Therefore, these workloads are the ideal beneficiary of multiple core processors. However, to fully exploit the speedup potential, algorithms should be carefully decomposed, reducing dependencies between kernels and data elements. In addition, shared hardware resources, such as memory and buses should be efficiently utilized and software overheads originating from thread scheduling and synchronization should be minimized.

In this paper we analyze the scalability of multithreaded multimedia workloads on state-of-art multiprocessors. We identify major scalability-limiting factors and illustrate how to address them. For our experiments, we carefully selected several representative multimedia applications, including video encoding/decoding, stitching of panoramic images and dense optical flow.

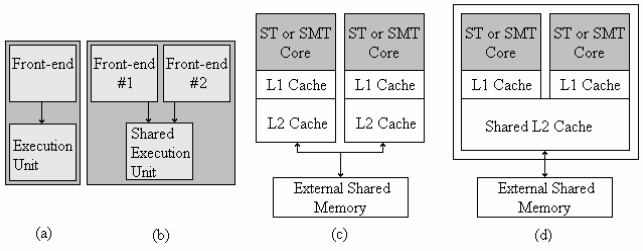

Figure 1: Block diagrams of different processors: (a) single-thread (ST) (b) *simultaneous multithreading* (SMT) (c) *symmetric multiprocessing* (SMP) (d) *chip-level multiprocessing* (CMP).

1.1. Background and hardware terminology

This section reviews the basic multi-core architectures, illustrated in Fig. 1. In Simultaneous Multithreading (SMT) a single physical core is partitioned into two or more logical cores. SMT allows multiple threads to execute instructions in the same clock cycle, thus enabling better utilization of the execution unit. Symmetric Multiprocessing (SMP) is a multiprocessor architecture in which identical processors are connected to a single shared off-die memory. As the shared memory is significantly slower than the processors, inter-thread communication through memory decreases the scalability. The shared memory bottleneck is reduced in Chip-Level Multiprocessing (CMP), namely multiple cores on a single die. Multiple on-die cores share a common cache, which typically has 10-times lower latency than an external memory.

Previous studies discussed the tradeoffs in implementing multithreaded software on actual processors. The performance of multithreaded scientific applications using SMT was described in [1]. Performance differences between SMP and SMT systems, and between SMT and CMP systems were reported in [2] and [3], respectively.

2. SCALABLITY OF MULTIMEDIA APPLICATIONS

In this study we focused on three representative applications that are likely to prosper in the coming years:

(1) H.264 video encoder that includes quantization, motion compensation, integer transform and entropy coding [4].
(2) Panoramic image generator that includes SIFT feature extraction, approximated nearest neighbor search and multiresolution image blending [5].
(3) Optical flow that consists of solving a set of linear equations using Successive Over-Relaxation (SOR) [6].

All applications were multithreaded using data decomposition, namely, partitioning of the data among threads, where each thread performs the same computation on different data, and synchronizes with the other threads if needed. We executed each application on 1-4 physical processors. Using accurate hardware timers, we measured the overall execution time, the sequential code time and the synchronization code time. Using VTune® performance analyzer, we measured the cache miss rates and the external bus utilization. The results are reported with respect to the execution time of a single thread on a single processor.

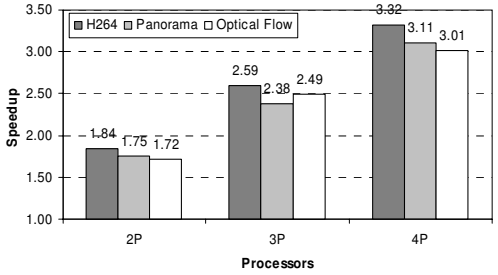

Figure 2: Speedups obtained on two (2P), three (3P) and four (4P) processors relative to the reference single processor code.

Fig. 2 shows the mean speedup values achieved on a multiprocessor Xeon® system with two, three and four physical processors, using diverse input data and operation parameters. Speedup values on two processors ranged from 1.6x to 1.9x, and on four processors from 2.7x to 3.4x.

The applications were tested on two multi-core systems: The first was an SMP system with four Intel 2.7GHz Xeon® processors, each having three levels of cache on-chip (L1–8KB, L2–512KB, L3-2MB), connected through a 400MHz front-side bus and running Windows® Server 2003 operating system. The second system was a CMP with an Intel 2.5GHz Core Duo® processor. Each core had a 32KB first-level cache, and both cores shared a 2MB second-level cache, and a 667MHz external memory bus. This system was running Windows® XP operating system.

3. SCALABILITY-LIMITING FACTORS

This section describes the dominant factors affecting the scalability results reported in the previous section. These include data access patterns, thread synchronization, operating system (OS) effects and algorithm design considerations.

3.1. Data access pattern

Sequential access to two-dimensional data is a common task in image processing algorithms. We have chosen to analyze a 2D Gaussian convolution, which consumes about 50% of the total execution time of the feature extraction module in our panoramic image construction application. However, the results apply also to other image manipulations, such as edge detection or noise filtering. Since low pass filtering is separable, it was implemented as two 1D phases: first, a horizontal 1x9 filter is moved across the rows, and then a vertical 9x1 filter is moved across the columns. Although this implementation closely follows the algorithm design, it does not take into account the actual representation of data in the memory system. Since the memory is one-dimensional, the horizontal filter achieves a good data locality when accessing data by the order of the rows, since most of the data is obtained from the fast cache (L1/L2). However, in the vertical convolution filter, accessing data by the order of the columns, the number of accesses to the slow external memory was about 30 times higher compared to the horizontal filter. As a direct consequence, the vertical filter executed 2.5 times slower than the horizontal filter. Furthermore, the inefficient data access had a negative effect on the scalability of the multithreaded implementation. The frequent attempts to access the shared memory increased the bus latency and reduced the scalability. As a result, while the horizontal filter showed a perfect speedup of 4x, the vertical filter executed only 2.4x faster on four processors, having 2-times higher average bus latency. A minor code optimization, scanning the vertical filter in row order, increased the scalability of the vertical filter from 2.4x to 3.9x, and the speedup of the entire module improved from 2.9x to 3.3x (Fig. 3).

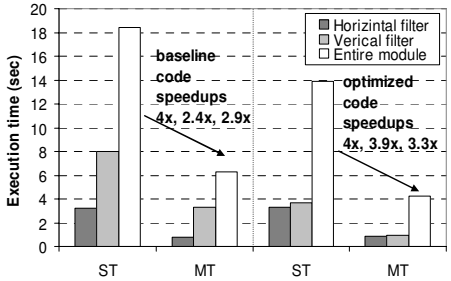

Figure 3: Execution time and speedups of single-thread (ST) and multithread (MT) versions of Gaussian convolution. Note that speedup factors improve for the optimized code.

3.2. Thread synchronization

Thread synchronization is a major source of overhead in multithreaded applications. In a previous work [4], we showed that synchronization by simultaneous access of four threads to lock-protected shared data structures can consume up to 35% of the total frame compression time in an H.264 video encoder. To reduce this overhead, we have designed and implemented a lock-free mechanism that manages the list of macroblocks available for processing. The mechanism

is based on atomic compare & swap (CAS) operations: Each thread reads a shared 64-bit data word, updates it and commits it through a CAS operation, retrying if the compare operation failed. This lock-free synchronization is general enough to handle shared task queues of other multithreaded algorithms as well. In the case of H.264 encoder, this mechanism significantly reduced the synchronization overhead and improved the application's scalability on four processors from 2.6x to 3.3x, as shown in Fig. 4. Similar improvement was achieved by a partial-lock policy [4] that allowed each thread to allocate some of its tasks locally, reducing the number of accesses to the lock-based shared data by factor 40.

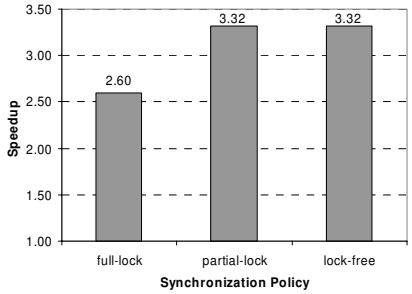

Figure 4: Speedup of an H.264 encoder on four processors, with full lock-based synchronization, compared to partial lock-based synchronization and lock-free synchronization

A second common type of thread synchronization is explicit event signaling, which is typically implemented by calling OS services. We measured the overhead of OS-based signaling in the multithreaded optical-flow application. Analysis of the scalability results showed that the speedup was limited by two factors: OS thread synchronization and thread imbalance. To improve the speedup, the OS synchronization was replaced by an efficient, assembly coded, synchronization function. Furthermore, as more threads were used and the workload was broken into smaller computational pieces, the load balancing improved as well, reducing the overhead of waiting for the last thread. These optimizations increased the speed up from 1.7x to 1.85x on two processors.

3.3. Memory management

Memory management is another basic service provided by the OS. This section emphasizes the importance of using thread-safe and scalable memory management libraries. In the following example, the multithreaded stitching module of the panorama application showed a very poor scalability of 1.4x on four processors (Fig. 5). Code analysis pointed out a single function, constituting about 25% of the entire execution time, which scaled down as the number of processors increased. This function utilized a linked list, implemented with standard template library (STL). The abstract list operations used frequent allocation and release of heap memory, which accessed the heap shared by all threads. The default heap library, provided by Windows OS, uses a lock-based mechanism to ensure thread safety: When a thread is accessing the heap, it first acquires a lock. If the lock is not available, the thread is suspended, waiting for the lock to be released. This mechanism enforces large overheads as the number of threads is increased, as it conceals a large number of implicit synchronization points. Using a lock-free heap library (LeapHeap, Necklace Ltd.) the problematic function became scalable, and the speedup of the stitching module on four processors was improved from 1.4x to 2.7x, as shown in Fig. 5.

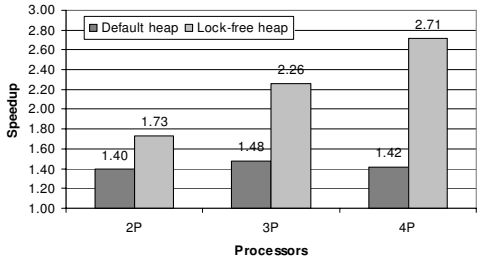

Figure 5: Speedups of the panorama stitching module obtained using the default heap and a lock-free heap.

3.4. Algorithm design

As described in previous sections, the developer of scalable multithreaded applications must consider factors related to the hardware architecture and the software infrastructure. However, the main challenge of achieving performance boost through thread-level parallelism remains in the domain of application design. The choice of a decomposition method suitable for the specific algorithm has an impact on the achieved scalability. The intuitive approach of partitioning the data symmetrically between threads might be sub-optimal due to algorithmic dependencies between data elements. These dependencies force the threads to use a synchronization mechanism, which has an additional run-time overhead. In addition, as the processing time of each data element is usually not constant, load balancing becomes an important consideration for achieving maximal system utilization. Multimedia applications commonly consist of computational kernels, wrapped with I/O and data pre-processing or post-processing code. This code is typically sequential, executed by a single thread. Consequently, the maximal achievable speedup is bounded, according to the well-known Amdahl's Law. Fig. 6 shows the potential speedup improvement in the H.264 encoder, if we were able to remove the sequential code and the algorithmic data dependencies. According to these estimations, a video encoder, designed with parallel execution in mind, would obtain an additional speedup improvement of 10%.

4. CHIP-MULTIPROCESSING ARCHITECTURES

This section compares the scalability achieved on CMP and SMP processor architectures. The speedup factors of all three applications on a CMP Intel Core Duo® processor, compared to a dual Xeon® SMP machine are shown in Fig. 7. Interestingly, CMP scalability was 3.5%-7% better than SMP.

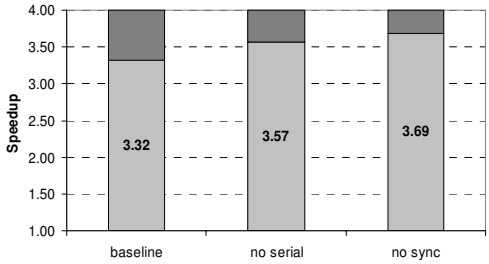

Figure 6: The achievable speedups of an H.264 encoder on four cores in the current design (baseline), when all serial code is removed (no serial), and when synchronization due to data dependencies is removed (no sync).

This scalability improvement results from the differences between the cache hierarchies of the two systems. In the CMP system (Fig. 1), the on-die L2 is shared by both cores. As a result, when multiple threads are working on adjacent inputs, good data locality is achieved for memory read operations. Moreover, shared data structures used for synchronization resides in L2, and write operations by one thread does not invalidate the data of the other threads. This is manifested by equal utilization of the external memory bus in single-thread and multithread executions. Conversely, in the SMP system the utilization of the external memory is higher when there are multiple threads, causing the scalability to reduce. Our measurements of the last-level cache misses in the H.264 encoder showed that while there was no difference between single-thread and multi-thread executions on the CMP system, on the SMP system the multithreaded encoder had 20% more cache misses than the single-thread encoder. The SMP performance remained the same even when the size of the last-level cache was increased from 512KB to 2MB, thus supporting our hypothesis that the improved scalability of the CMP system is due to the shared cache architecture.

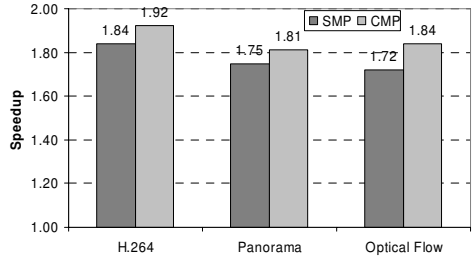

Figure 7: Speedups achieved on CMP and SMP systems

5. DISCUSSION AND SUMMARY

Over the past decade, the rapid increase in computation power of commodity PCs provided an impressive performance boost to multimedia applications, with relatively low effort on the part of the application developers. This is about to change, as multi-core platforms are expected to dominate PC systems. By using thread-level parallelism on these platforms, developers of multimedia applications can gain a significant performance improvement, far beyond the usual gain of transitioning to a new processor model. However, full utilization of multi-core processors requires application developers to be aware of the hardware architecture, memory organization, and OS mechanisms. Furthermore, it requires algorithm designers to remodel their algorithms, having parallel execution in mind. As demonstrated in Section 3, achieving high speedups by multithreading existing application code is a challenging task. Fortunately, once an application is designed for good scalability, it will profit effortless further improvement as the number of cores per processor grows.

The following guidelines summarize our findings:

(1) Good utilization of local cache is required to minimize the accesses to shared memory and reduce bus latencies.
(2) Optimized single-thread performance is beneficial for improving multithread performance.
(3) The number of synchronization points between threads should be minimal. Lock-based synchronization should be avoided as much as possible.
(4) Optimized synchronization primitives should be preferred over current OS services.
(5) Frequent memory allocation and release should be avoided unless the shared heaps are lock free.
(6) Application design should minimize the portions of sequential code and the dependencies between data elements, allowing straightforward parallelization..

In conclusion, this study observed major scalability bottlenecks in various multimedia applications. We presume that these applications provide a figure of merit for the achievable speedup of other multimedia applications as well. As the number of cores per processor is expected to continually grow in the foreseeable future, these scalability issues are expected to intensify, and further work is required to consolidate our observations on larger-scale systems.

6. ACKNOWLEDGEMENTS

We would like to convey our gratitude to M. Tsadik, Y. Kulbak and S. Yefet for their contributions to this research.

7. REFERENCES

[1] R. Grant and A. Afsahi, "Characterization of Multithreaded Scientific Workloads on Simultaneous Multithreading Intel Processors," IOSCA 2005, Austin, TX, USA, 2005, pp. 13-19.

[2] Y.K. Chen, R. Lienhart, E. Debes, M. Holliman, M. Yeung. "The Impact of SMT/SMP Designs on Multimedia Software Engineering — A Workload Analysis Study," MSE 2002, pp. 336.

[3] C. Liao, Z. Liu, L. Huang, and B. Chapman, "Evaluating OpenMP on Chip Multithreading Platforms," University of Houston, USA, July 2005.

[4] G. Amit and A. Pinhas, "Real-Time H.264 Encoding by Thread-Level Parallelism: Gains and Pitfalls", PDCS 2005.

[5] M. Brown and D. G. Lowe. "Recognising Panoramas", ICCV2003, pages 1218-1225.

[6] M. Black and P. Anandan, "A framework for the robust estimation of optical flow", ICCV-93, May, 1993, pp. 231-236.