SCALABILITY OF MULTIMEDIA APPLICATIONS ON NEXT-GENERATION PROCESSORS

Guy Amit, Yaron Caspi, Ran Vitale, Uri C. Weiser and Adi T. Pinhas
Corporate Technology Group, Intel Corp, 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. The paper lists guidelines for proper utilization of these new multi-core platforms. In particular, the study discusses 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 discuss 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 parallelism in the program. One approach is *Instruction-Level Parallelism* (ILP), which are instructions that can be executed concurrently on several execution units. Another approach gained speedup from *Data-Level Parallelism* (DLP), which allows executing a specific operation on multiple data elements within a single instruction (e.g. MMX). Recent processors support *Thread Level Parallelism* (TLP) by running two or more threads simultaneously on multiple physical or logical cores. Future processors are expected to have a larger number of cores on die. The architectures of these new technologies are illustrated in Fig. 1.

This paper studies the expected speedup that these architectures can provide for high-performance applications. Multimedia workloads are the ideal beneficiary of multiple core processors, having independent kernels and steady computation patterns that enable functional decomposition. 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.

This paper analyzes the scalability of multithreaded multimedia workloads on state-of-art multiprocessors. We identify major scalability-limiting factors and illustrate how to address them. Our suggestions may be useful for algorithm designers as well as for application developers. For our experiments, we carefully selected several representative multimedia applications, including video encoding/decoding, stitching of panoramic images and dense optical flow.

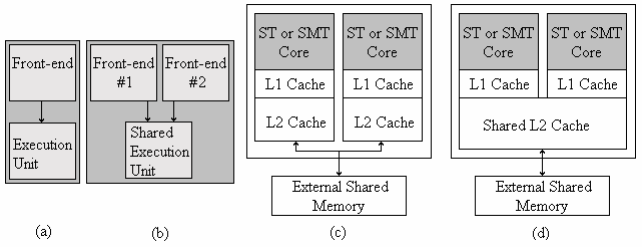

Figure 1: Block diagrams of a single-thread processor (a), a SMT processor (b), SMP processor (c) and CMP processor (d).

1.1. Background and hardware terminology

This section briefly reviews the basic multi-core architectures, illustrated in Fig. 1. Symmetric Multiprocessing (SMP) is a multiprocessor computer architecture in which identical processors are connected to a single shared off-chip memory. As the shared memory is significantly slower than the processors, inter-thread communication using the memory will decrease the scalability. The shared memory bottleneck is reduced in Chip-Level Multiprocessing (CMP), which is SMP implemented on a die. Multiple on-die processor cores share a common cache, typically 10-times faster than an external memory. 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.

Few 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 over 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 a SIFT feature detector, approximated nearest neighbor search in a KD-tree and multiresolution spline image blending [5].
(3) Optical flow [6] that consists of solving a set of linear equations using Successive Over-Relaxation (SOR).

All applications were multithreaded using data decomposition, namely, partitioning of the data between the threads, where each thread performs the same computation on different data, and synchronizes with the other threads if needed. We executed each of the applications 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 processor, i.e., the execution times of a single thread on a single processor were defined as a reference line (scalability = 1).

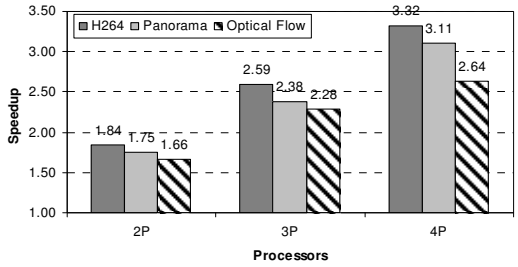

Figure 2: The obtained speedups using platforms with two (2P), three (3P) and four (4P) processors with respect to the reference code running on a single processor.

Figure 2 shows the mean speedup values achieved on a multiprocessor Xeon® system with two, three and four physical processors, using different 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 multithreaded applications were tested on two multi-core systems: The first was an MP system with four Intel 2.7Ghz Xeon® processors, each with three levels of cache on-chip (L1 – 8KB, L2 – 512KB, L3 - 2MB), connected through a 400Mhz front-side bus (100Mhz quad data rate) and running Windows® Server 2003 operating system. The second system was a CMP with an Intel 2.5Ghz Core Duo®. Each core has a 32KB first-level cache, and both cores share a 512KB second-level cache, and a 667MHz bus to the external memory. This system was running Windows® XP operating system.

3. SCALABILITY-LIMITING FACTORS

This section describes the dominant factors that are crucial for obtaining 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 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, a good data locality is achieved when accessing data in the order of the rows, and most of the data is obtained from the fast cache (L1/L2). However, when accessing data in the order of the columns, the number of accesses to the slow external memory was about 30 times higher in the vertical filter as compared to the horizontal filter. As a direct consequence, the vertical filter executed 2.5 times more slowly than the horizontal filter. Furthermore, inefficient data access had a negative effect on multithread scalability. 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 (scan 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 (Figure 3).

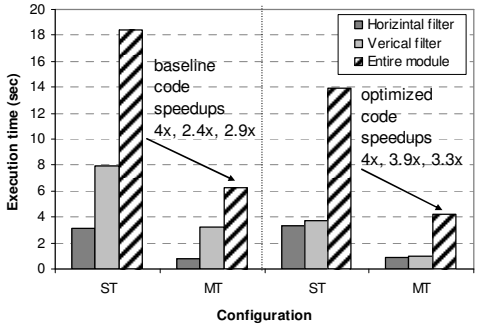

Figure 3: Execution time of single-thread (ST) and multithread (MT) versions of horizontal and vertical Gaussian convolution (part of SIFT module of the panorama application). Speedups are shown for the baseline and optimized versions.

3.2. Thread synchronization

Thread synchronization is a major source of overhead in multithreaded applications. In a previous paper [4], we showed that synchronization by simultaneous access 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 significantly mechanism reduced the synchronization overhead, and as shown in Figure 4, improved the application's scalability on four processors from 2.6x to 3.3x. Similar improvement was achieved by a partial-access policy that reduced the number of accesses to the lock-based shared data by factor 40.

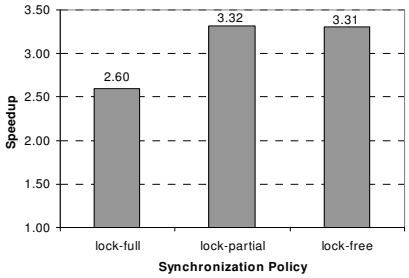

Figure 4: Speedup of H.264 encoder on four processors, with full lock-based synchronization, compared to partial lock-based synchronization and lock-free synchronization

A second common type of thread synchronization is implicit event signaling, which is typically implemented by calling OS services. To measure the overhead of using the OS signaling, we used an optical-flow application. Analysis of speedup results showed that the speedup was limited by two equally weighted factors: thread imbalance and OS threads synchronization. To improve the speedup, the OS synchronization mechanism was replaced by an efficient synchronization function, coded in assembly. This improved the speed up from 1.7x to 1.85x on two processors. Furthermore, as the number of threads increased, the thread load balancing improved as well, and the total speedup was better than linear with the number of threads.

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 (Figure 5). Code analysis pointed out a single function, constituting about 25% of the entire module 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 use frequent allocation and release of heap memory, which access the heap shared by all threads. The default shared 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 locking 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 was improved from 1.4x to 2.7x, as shown in Figure 5.

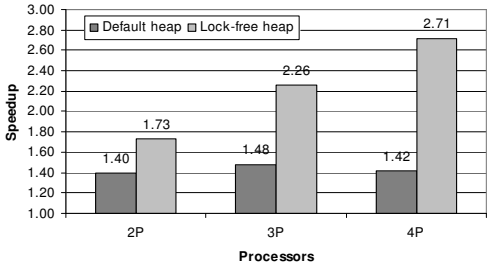

Figure 5: Speedup values obtained for the panorama application stitch module using the default NT heap compared to a lock-free heap.

3.4. Algorithm design

As described in previous sections, the developer of scalable multithreaded application must take into consideration 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 a major impact on the achieved scalability. The intuitive approach of partitioning the data symmetrically between threads might not be 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 utilization of the system. 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. Figure 6 shows the potential speedup improvement in the H.264 encoder, if the algorithm designers were able to remove the scalability limitations imposed by the sequential code and by the algorithmic data dependencies. According to these estimations, a video encoding algorithm, designed with

with parallel execution in mind, could have obtained additional 10% in speedup.

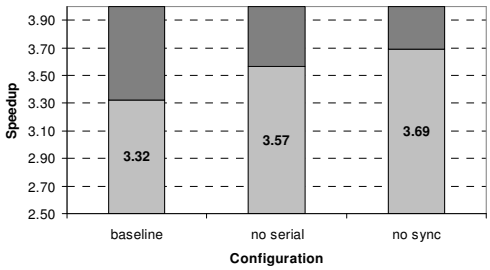

Figure 6: Potential scalability. The achievable speedups of H.264 encoder on four cores in the current design (left), when serial code is removed (middle), and when synchronization due to data dependencies is removed (right).

4. CHIP-MULTIPROCESSING ARCHITECTURES

This section describes the differences in scalability between two different processor architectures: CMP and SMP. The speedup factors of all three applications on a CMP Intel Core Duo® processor, compared to a dual Xeon® SMP machine are shown in Figure 7. CMP scalability is 4%-7% better than SMP. This scalability improvement is due to the differences between the cache hierarchies of the two systems. In the CMP system, the on-die L2 is shared by both cores. As a result, L2 data is not invalidated when the OS migrates threads between cores. Moreover, the shared data resides on L2, and it is not required to access the external memory whenever one thread updates the data. This is manifested by equal utilization of the external memory bus in single-thread and multithread executions. In the SMP system, L2 data becomes invalidated more often when there are multiple threads, and the utilization of the external memory is higher than the single-thread execution, causing the scalability to reduce.

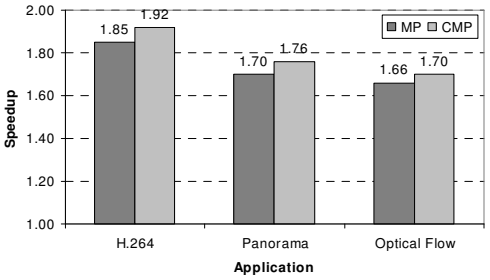

Figure 7: Speedups achieved on CMP and MP systems

5. DISCUSSION AND SUMMARY

Over the past decade, the rapid increase in computation power of commodity PCs has given an impressing 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 fine details 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 good scalability by multithreading existing code is a challenging task, but once an application is designed for good scalability, it will profit effortless further improvement as the number of cores per processor grows.

The following summarizes the guidelines derived from our analysis in the previous sections:

(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 crucial 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 OS services.
(5) Frequent memory allocation and release should be avoided unless the shared heaps are lock free.
(6) Application design should be 'parallel', minimizing the portions of sequential code and the dependencies between data elements.

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.