White Paper

Intel Corporation

Prefetching

Optimizing Software for Multi-core Processors

TABLE OF CONTENTS

Executive Summary 2

Assessing Performance Optimization Opportunities 3

AMIDE 3

Analyzing Existing Code 4

Evaluating Various Methods to Parallelize Code (Functional vs. Data Parallelism) 8

Measuring Code Performance 12

Conclusion 14

Appendix 1: Software Analysis Tools 15

Appendix 2: Hardware Configuration 15

Executive Summary

How much software optimization is enough? This question is even more perplexing with multi-core processors whose additional cores may require some effort to gain maximum performance benefits. Multi-core processors offer a refreshing approach for developers who want to improve overall system performance; however, there may be new software issues to deal with. This paper is intended for software developers who want a template for identifying performance optimization opportunities based on a real world application.

We will present the code migration of an open-source medical image rendering application, a serial to multithreaded transformation. This application was threaded to take advantage of a four-core system (two dual-core processors). After exploring alternative schemes for parallelizing the application, a popular thread library called pthreads (POSIX* Portable Operating System Interface) was used. We studied five performance metrics and concluded the resulting code was well-optimized, demonstrating a 3.3X performance improvement compared to single-core implementations[+].

Assessing Performance Optimization Opportunities

The process to migrate software from single-core to multi-core processors can be broken down into three major exercises. First, software developers analyze the "starting" application software to understand the code's intrinsic parallelism and how efficiently it uses hardware resources. Next, developers evaluate the various methods to parallelize their application and implement the most suitable method. Once threaded code is in place, it must be tested for correctness to determine whether any errors in computation were introduced by threading the application. Finally, developers measure code performance using analysis tools which pinpoint bottlenecks, stalled processes or poor hardware resource utilization. This evaluation employed Intel® performance tuning and analysis tools which are described in Appendix 1.

We applied this three-step approach to an image rendering application called AMIDE*, enabling it to run efficiently on two dual-core processors as shown in Appendix 2. Using rendering techniques, AMIDE operates on medical imaging data sets containing geometry, viewpoint, texture and lighting information and generates an image for display on a PC monitor.

AMIDE*

AMIDE (A Medical Imaging Data Examiner) is a free, open-source tool for viewing. The following URL contains more information on AMIDE: http://AMIDE.sourceforge.net. AMIDE supports a range of features including some basic commands for rotating images, selecting layers to view, and fusing multiple images. A screen shot from AMIDE is shown in Figure 1.

Our parallelization efforts[1] concentrated on the underlying image rendering engine. AMIDE utilizes a library called VolPack to perform this function[2]. VolPack is a portable software library written by Philippe Lacroute, and the software is covered by the Stanford Computer Graphics Laboratory's General Software License[3].

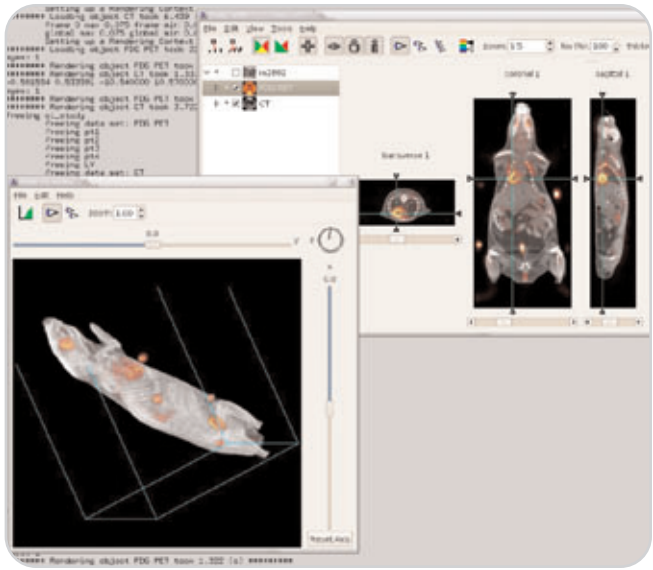


Figure 1. Image rendering by AMIDE*

Analyzing Existing Code

We employed a six-step approach to analyze AMIDE before porting it to a multi-core processor system.

1. Optimize the Starting Serial Code

Before beginning code parallelization, it is imperative to optimize poorly performing sections of the application. The largest gains come from parallelizing the portions of the code that are executed most often and consume the greatest amount of execution time. If, through serial optimization, the percentage of time spent in different function calls changes, then the decision on which functions to parallelize may change as well. Once the code is optimized for the sequential case, a performance profiling tool should be used to determine the best candidates for parallelization.

2. Gather Information about the Code

The second step in analyzing existing code is to gather information about the run-time performance and the control characteristics of the code. The former indicates code segments that will benefit most from the parallelization; the latter identifies specific code locations to split the work. This analysis highlights the code's intrinsic parallelism and indicates how well the hardware resources, such as processor cache memory, will handle the data structures.

Determining the run-time performance gain requires a set of performance metrics to measure before (single-core) and after (multi-core) results. We instrumented the code with timers to calculate actual time required to render images.

For those with a prior working knowledge of the software architecture of the targeted application, this information gathering step may require only little incremental effort. In our case, we were unfamiliar with AMIDE, so we had to study the code to determine the dataflow and the data structures. Our analysis of AMIDE and the VolPack library resulted in the data flow diagram shown in Figure 2. This data flow depicts the flow of data through major code blocks and three loops, and represents the software architecture of the application.

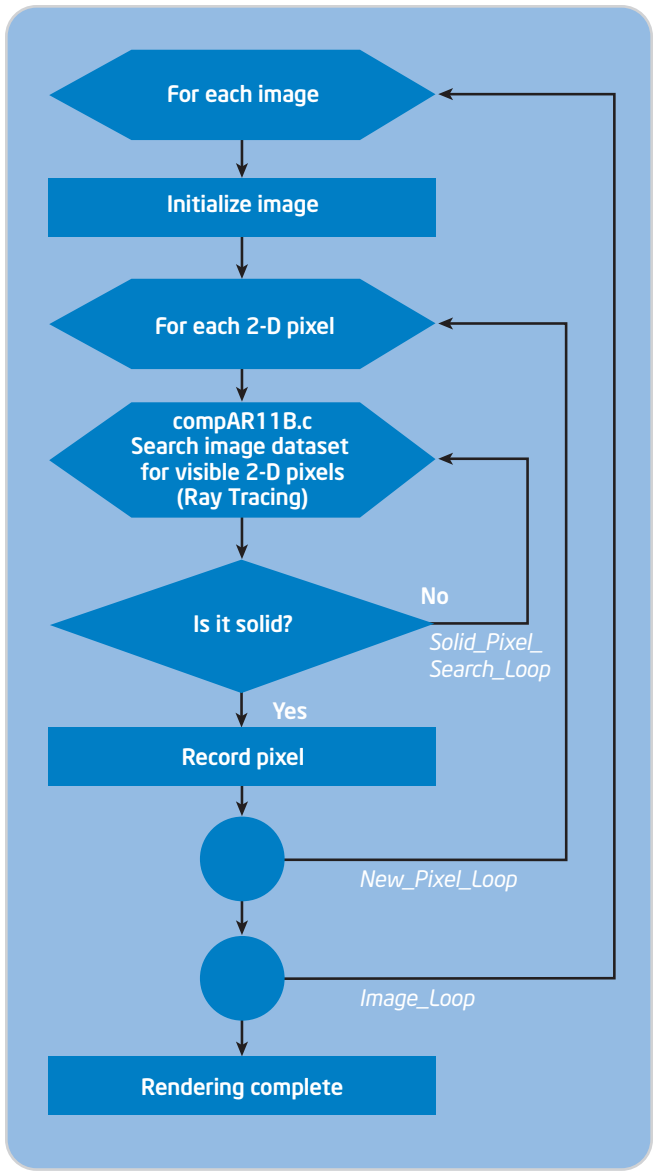


Figure 2. AMIDE/VolPack Data Flow Diagram

The following is a basic description of the data flow.

a. The user loads an image, and the AMIDE application initializes this image.

b. The application generates a 2-D display image for the monitor. For each 2-D pixel, the application searches the image dataset stored in an array and displays either a portion of the medical image or a white pixel. For each pixel that will be displayed on the screen, VolPack runs a loop called the New_Pixel_Loop.

c. There is a nested loop within the New_Pixel_Loop called the Solid_Pixel_Search_Loop. This loop uses Ray Tracing techniques to search the medical imaging dataset to determine whether the pixel should be displayed to the user. Ray Tracing is an image rendering technique, based on complex mathematical algorithms, to model the reflections and refractions of light incident to surfaces and materials. Once Ray Tracing finds the pixel to display, it is recorded and the Solid_Pixel_Search_Loop terminates.

d. The New_Pixel_Loop terminates when all the pixels have been determined.

e. The Image_Loop processes multiple images loaded by the user. This loop terminates when all the loaded images have been rendered.

After identifying the major code blocks from the dataflow analysis, we used a code analysis tool (Intel® VTune™ Performance Analyzer) to determine how much time is spent in each block. If a tool is not available to make execution time measurements, the code can be instrumented with hardware timers to measure the time spent in each major block. The measurements indicated the majority of the work was done in the compAR11B.c block shown in Figure 2 on the previous page, and we determined that the loop that calls compAR11B.c was the best candidate for parallelization.

We collected AMIDE performance data as measured by the time required to render an image after the user requests to view the medical image from a new angle. Depending upon the angle of rotation, AMIDE decides whether to search the medical image dataset from the X-, Y- or Z-axis. We found the time required to display an image was dependent upon the axis AMIDE selected to traverse. For example, a search along the Z-axis tends to be roughly three times faster than along the X-axis, illustrated for 1000 rotations in Figure 3. We used image rendering time as our performance metric to measure the effectiveness of our parallelization implementation.

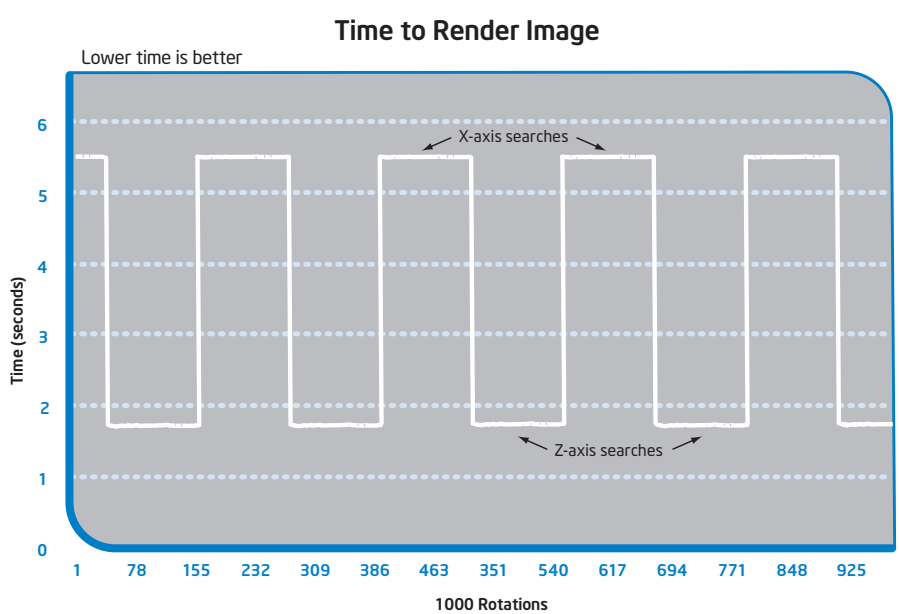


Figure 3. Time to render an image for 1000 rotations

This X- and Z-axis render time variance is explained by the memory stride of the 2-D pixel search as represented in the AMIDE application. For the Z-axis, bytes are accessed sequentially with a stride of four bytes, which is a relatively short stride. For the X-axis, accesses have a stride of 976 bytes which is relatively long and results in more memory paging overhead. For more information on the performance implications of strided memory accesses, please see the side-bar below, which includes a link to another White Paper that discusses this topic.

3. Pinpoint External Dependencies

The third step in analyzing existing code is to inspect the code to see whether there is a high degree of idle time. When idle time is minimized, the functions consuming the most time tend to be compute-intensive pieces of the code. If code functions are waiting for something – the hard drive, network, user input, etc., parallelization will not speed up those external interfaces. However, parallelization is sometimes used to make multiple, simultaneous accesses to I/O devices, such as hard disk drives, to increase the aggregate data transfer rate.

We determined that the code in question (compAR11B.c in Figure 2 on page 4) was free of external dependencies, ran without interruption, and had no interaction with peripheral devices. These attributes supported the decision to around parallelize this portion of the code.

4. Identify Parallelization Opportunities

The fourth step is to decide how to parallelize the code. We can choose from two methods: dividing the data or dividing the control. If we parallelize by data, each of the four cores would be responsible for accessing pixels in its quarter of the array. To make this work, the array indexes can simply be divided into four equal-sized ranges and assigned to individual cores. On the other hand, if we parallelize by control, one of the loops would be spilt by its iterations and each core would process a quarter of the loops. For many applications, these two methods are often equivalent because the same loop iteration accesses the same data each time this function is called. However in our case, a render does not necessary touch all the data and dividing the computation by data could lead to great load imbalances. Instead, we split the computation by the control. Each core was responsible for determining a quarter of the 2-D pixels that would be displayed.

In order to split by control, we had to decide at what level to parallelize this code. In other words, if the code is nested, how deeply within the nesting structure should the threads be created? And which loop should be split among the four cores? (see Figure 2) These decisions depend on both performance testing and a good architectural understanding of the code. We evaluated three nesting levels corresponding to the three loops (lowest to highest) shown in Figure 2: Solid_Pixel_Search_Loop, New_Pixel_Loop, Image_Loop.

Each time a thread begins, there is overhead associated with starting it. Likewise, communicating final results, as in sharing data, is costly. Because of the synchronization and consequent serialization inherent in communication between threads, it is desirable to have the threads perform the most work with the least amount of coordination.

The lowest threading level, compAR11B.c contains the function with the greatest workload which runs the Solid_Pixel_Search_Loop. Within this loop, compAR11B.c accesses the main array and checks for pixels containing data. It stops when it finds a pixel with data. This type of loop is not appropriate for parallelization because, for any given execution, it executes for an undetermined number of iterations. This loop could consume one core while the other cores are idle, waiting for a synchronization event to occur.

The next level kicks off the Initialize Image function that runs the New_Pixel_Loop, with a deterministic number of iterations (e.g., while it is not a constant number, the number is known at the beginning of the loop execution). This loop calls compAR11B.c during every iteration. This loop is an excellent candidate for parallelization because we can divide the iterations by the number of threads and

For some applications, such as packet processing and medical imaging, we have seen performance improvements when the hardware-assisted data prefetching is turned off. This is because the prefetcher may have difficulty with applications with long-stride data accesses (on the order of 1 Kbytes or more) as in medical imaging or with accessing data in a random fashion as in packet processing. We have collected AMIDE performance data for three data prefetching schemes: with hardware prefetch, without hardware prefetch and with software prefetch instructions. This information is presented in a companion white paper called "Optimizing Embedded System Performance – Impact of Data Prefetching on a Medical Imaging Application". Please see http://www.intel.com/design/embedded/papers/315256.htm for more details.

distribute the workload evenly among the four cores. Another benefit from parallelizing this "for"' loop is that, since the threads will call compAR11B.c multiple times before coordinating with each other, synchronization overhead will be minimized.

At the highest level, the Image_Loop could be parallelized so each thread renders a different image. AMIDE would then call this function multiple times, once for each image, displaying multiple images at the same time. This has two possible drawbacks. The first is there will be idle cores whenever the number of images being processed is less than the number of cores. The second is that each image requires substantially different amounts of time to render. If one thread completes its image much earlier than the other, there is parallelism for only part of the time. Thus, load balancing is less consistent when the threads are split at this higher level.

5. Locate Data Dependencies

The fifth step in analyzing the existing code is examining the data structures to assess whether parallelism may create data dependencies, causing incorrect execution or negatively impacting performance. One source of data dependency occurs when multiple threads write to the same data element. Without additional synchronization code, this can produce incorrect results.

In addition to ensuring that multiple threads are not writing to the same data element, it is important to minimize any false sharing of memory among the threads. False sharing occurs when two or more cores are updating different bytes of memory that happen to be located on the same cache line. Technically, the threads are not sharing the exact memory location, but because the processor operates on cache lines (on the order of 64 bytes) which span several memory locations, the bytes end up getting shared anyway. Since multiple cores cannot cache the same line of memory at the same time, the shared cache line is continually sent back and forth between cores, creating cache misses and potentially large memory latencies.

AMIDE calls the VolPack library to render a 2-D display image by processing a medical image dataset. This requires searching the medical image dataset, but never changing it, and writing the output as a 2-D display image to a graphics driver for display on a monitor. Both structures, the medical image dataset and the 2-D display image, are stored as arrays.

Since the 2-D display image is modified by all the cores, we need to guarantee that no two cores write to the same or neighboring locations in the array at the same time. Two cores writing to neighboring locations (false sharing) at the same time may result in cache inefficiencies due to shared data, whereas writing to the same location necessitates costly synchronization code.

Each loop iteration corresponds to one location (pixel) in the 2-D image, and consecutive iterations write to neighboring locations. Thus, by giving each thread a set of consecutive iterations, rather than assigning the iterations round-robin, the cores never write to the same location, and they only write to neighboring locations on their first or last iteration.

6. Measure System Resource Utilization

The sixth and final step is measuring the code's utilization of hardware resources such as CPU core, cache memory and I/O. These metrics can be collected by code analysis tools that measure CPU core utilization and cache hit rates. CPU core utilization indicates how much performance headroom is available.

Running the cores at 70-90% utilization can provide some compute slack time to handle peak loads; however, lower core utilization may indicate workload imbalances. Cache hit rate measurements indicate how often the CPU accesses instructions and data elements from high speed cache versus slower system memory (about 80x slower).

VolPack's image generation is basically a compute-driven and I/O-independent operation which consumes nearly 100% of a single core CPU. Therefore, the CPU is well utilized.

Running on one core, the L2 cache hit rate was 78% when the 2-D pixels were searched along the Z-axis. This means the CPU execution unit was able to get its instructions and data from the high speed cache a vast majority of the time. Cache hit rates above 90% typically indicate well-optimized software. Conversely, when the 2-D pixels were searched along the X-axis, the cache hit rate was only 26%.

Compared to pixel searches along the X-axis, Z-axis searches are about 3 times faster and the cache hit rates are about three times greater. It's clear the medical image dataset, a one dimensional array, is not optimized for all three axes.

One reason for this relatively high cache hit rate along the Z-axis is the relatively short-stride memory access (4 bytes). In other words, AMIDE reads regular, sequential data blocks that fit nicely into cache memory. This data regularity also allows the CPU to more easily predict the data required for future instructions. Based on these predictions, the CPU preemptively moves data from memory into its cache to hide memory latency. This process is called "prefetching" because the CPU preloads its cache in anticipation of future accesses.

In contrast, the cache hit rate for searches along the X-axis was 26% and this is associated with the relative long memory stride (976 bytes). With memory access strides of 1KB or greater, the data prefetcher may become significantly less effective. In order to improve the performance along the X-axis, the data structure would need to be redesigned to reduce the memory access stride for X-axis searches; this would be a major software development effort.

Evaluating Various Methods to Parallelize Code (Functional vs. Data Parallelism)

Dataflow and data structure evaluations on the starting code provide a feel for the parallelization opportunities. In particular, the dataflow is helpful in determining whether the application exhibits intrinsic functional or data parallelism. This understanding helps evaluate different code parallelization methods.

- When there are a number of independent tasks that run in parallel, the application is suited to functional decomposition. Explicit threading is usually best for functional decomposition.
- When there is a large set of independent data that is processed through the same operation, the application is suited to data decomposition. Compiler directed methods, such as OpenMP*, are designed to express data parallelism. Thread libraries, such as POSIX threads (Pthreads) developed for Linux, UNIX* and other operating systems, are also useful. For a shared-memory system, OpenMP and Pthreads are the most common packages.

An examination of the AMIDE/VolPack dataflow shows that most of the work is performed by the Solid_Pixel_Search_Loop, which is run on all 2-D pixels, so this application has inherent data parallelism. Since 97% of the work is done by one function in this loop (comp-AR11B.c), as opposed to multiple serial functions, this application is not a strong contender for functional parallelism.

For either functional or data parallelism, the programmer may write explicit threads to instruct the operating system to run these tasks concurrently. An explicit thread is comprised of purposely coded instructions using thread libraries such as POSIX threads or Win32* threading APIs. Programmers are responsible for creating threads manually by encapsulating independent work into functions that are mapped to threads. POSIX threads give the programmer complete control over what thread performs what work. However, it is more difficult to program because the programmer is responsible for all synchronization, the starting and stopping of threads, and special code that assigns the work to each thread. Like memory allocation, thread creation must also be validated by the programmer.

Implicit Threading Using OpenMP

Although explicit threads are general purpose and powerful, their complexity may make compiler directed threading a more appealing alternative. An example of compiler directed threading is OpenMP which is an industry standard set of compiler directives introduced in 1997. In OpenMP, programmers use pragmas to describe parallelism to the compiler, e.g.:

```
#pragma omp parallel for private(pixelX,pixelY) for (pixelX = 0; pixelX < imageHeight; pixelX++)
{
  for (pixelY = 0; pixelY < imageWidth; pixelY++)
  {
    newImage[pixelX,pixelY] =
             ProcessPixel (pixelX, pixelY, image);

    }
}
```

The "pragma omp" indicates this is an opportunity for OpenMP parallelism. The "parallel" key word tells the compiler to create threads. The "private" clause lists variables that need to be kept private for each thread to avoid race conditions and data corruption. The "for" key word tells the compiler the iterations of the next for-loop will be divided among those threads.

The compiler creates the spawned threads as shown in Figure 4. Notice the spawned threads are all created and retired at the same time somewhat resembling the tines of a fork. This is called a fork-join model and is a required characteristic for OpenMP parallelism. OpenMP pragmas are less general than threaded libraries, but they are less complex to program since the compiler creates the underlying parallel code for the multiple threads. OpenMP is supported by various compilers allowing the threaded code to be transportable, whereas threaded libraries typically have allegiance to specific operating systems.

OpenMP is most appropriate if there are loops to parallelize, and all of the loop iterations are entirely independent. It is very important that there are no data dependencies between them. Similarly, "false sharing," as discussed previously, should be avoided. Thus, in this case, it is important that consecutive iterations are assigned to the same core.

Synchronization

Parallelization often requires synchronization between threads. Synchronization occurs when threads are started and stopped or whenever they read or write to a shared data structure, so we will address these separately. First, it is important to reduce the number of starts and stops of threads. The desire is to parallelize with as much work as possible for each thread, while at the same time making sure that every thread works for approximately the same amount of time.

Second, in order to reduce writes to shared data structures, threads should write to local variables for as long as other threads do not need the data. For example, when summing array elements, the threads should keep a local tally and only add their sum to the global sum after they have completed their work. This way, they will get cache hits all the way until the final update of the global sum.

Parallelizing AMIDE/VolPack

VolPack performs the same operation on each pixel, and this characteristic combined with the relatively simple data structure provides for a straight forward parallelization strategy. We used POSIX threads to divide the New_Pixel_Loop into four sub-loops, each running on one of four threads.

In our implementation, Thread 1 first executes serial code to load the image which precedes the dataflow shown in Figure 2 on page 4. Next, Thread 1 uses POSIX threads to spawn three threads 2-4.

We considered using a fork instead of POSIX threads, but this option was dismissed because the fork system call does not share memory between processes. Although memory can be shared between processes using a Message Passing Interface (MPI), it runs slower than a shared memory model.

Two professors from the California Polytechnical State University took two days to inspect the code, parallelize the image rendering code, and test the workload balance. The effort to parallelize other applications will depend on the nature of the starting code and the parallelization experience of the software developers.

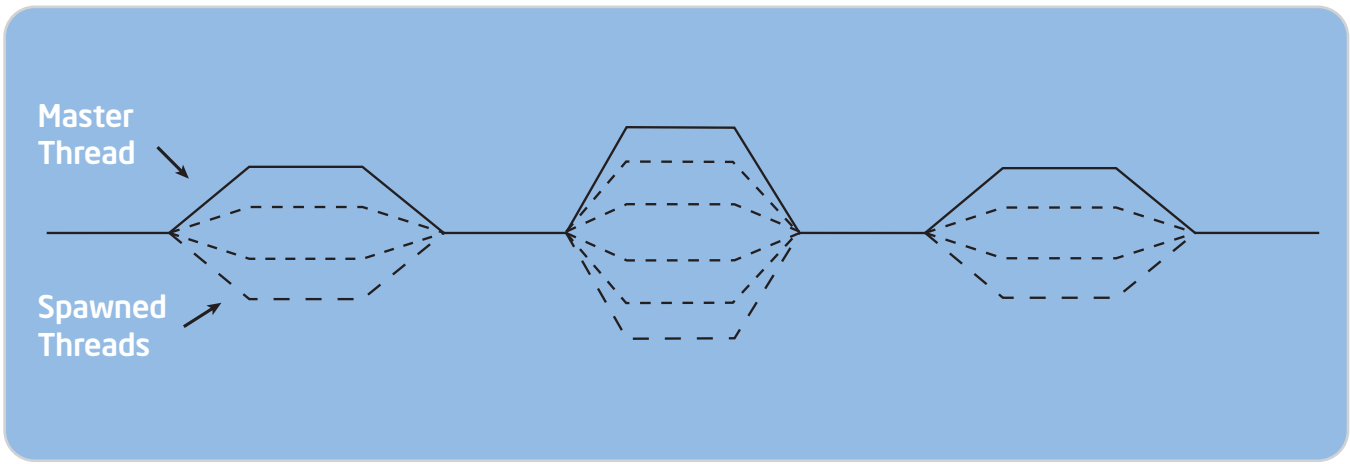


Figure 4. Fork-Join Model

```
{
      //printf("Start all of our threads!!!!\n");
            vp_begin_threads();
            vp_threads_begun = 1;
}

// allocating and spawning the threads
void vp_begin_threads()
{
            int i;
            int mask = 0xf;
            vp_pthreads_data = vp_create_thread_data();
            if (vp_pthreads_data == NULL)
            {
                  printf("Unable to allocate memory for threads.\n");
                  return;
            }
            // thread 0 is the master thread – spawn the others now
            for(i=1;i<NUM_THREADS;i++)
            {
                  vp_pthreads_args[i].data = vp_pthreads_data;
                  vp_pthreads_args[i].id = i;
                  pthread_create(&(vp_threads[i]), NULL, vp_thread_task_loop,
                              &(vp_pthreads_args[i]));
            }
}
```

Creating Threads

The following code segment initializes a number of threads (NUM_THREADS) using the pthread_create command. Please note that Pthreads is a commonly used abbreviation for POSIX threads. Each thread will execute the same function, vp_thread_task_loop.

The preceding code segment uses a variable NUM_THREADS which corresponds to the number of cores in the system. Since the number of cores is stored in a defined symbol, it can be easily ported to run on systems with any number of cores.

Starting Threads

After the threads have been created, Thread 1 parallelizes AMIDE at the New_Pixel_Loop to run on four cores. The following code segment, located in the threaded function vp_thread_task_loop, illustrates how this is accomplished.

```
vp_thread_big_loop_args loop_args[NUM_THREADS];
            int num = (kcount)>>THREAD_SHIFT;
            int extras = (kcount)&THREAD_MASK;
     int cur_num = kstart;

     // first set up the values that are the same for everyone
     loop_args[0].vpc = vpc;
     loop_args[0].kstart = kstart;
     loop_args[0].kinc = kinc;
     loop_args[0].icount = icount;
     loop_args[0].jcount = jcount;
     loop_args[0].kcount = kcount;
     loop_args[0].istride = istride;
     loop_args[0].jstride = jstride;
     loop_args[0].kstride = kstride;
     loop_args[0].composite_func = composite_func;

     for(i=0;i<NUM_THREADS;i++)
     {
            loop_args[i] = loop_args[0];  // copy the original arguments
            loop_args[i].kmystart = cur_num;  // record my starting loop index
            loop_args[i].id = i;
            cur_num += (num * kincr);
            cur_num += ((i < extras) * kincr);
            loop_args[i].kstop = cur_num;    // record my ending loop index
     }

     // start each thread
     vp_pthreads_data->completed_threads = 0;
     for(i=1;i<NUM_THREADS;i++)
     {
            vp_pthreads_data->inputs[i] = &(loop_args[i]);
            vp_pthreads_data->task_number = LOOP_TASK;
            pthread_cond_signal(vp_pthreads_data->task_cond[i]);
    }
```

Each thread is assigned variables in a global array using the following instruction.

loop_args[i].kmystart = cur_num

One quarter of the array indexes are passed to each thread to process the image volume using the variable loop_args[i]. The four threads are started by first assigning memory for data with the following command:

vp_pthreads_data->inputs[i] = &(loop_args[i])

Next each thread begins executing the LOOP_TASK code initiated by the following instruction:

vp_pthreads_data->task_number = LOOP_TASK

Finally, each thread is released to begin processing using the instruction:

pthread_cond_signal(vp_pthreads_data->task_cond[i])

For some applications, we have seen supra-linear performance improvement when migrating to multi-core processors. For example, migrating SNORT*, an open source intrusion detection application, from a single-core system to a four-core system resulted in more than a six times performance improvement. By employing particular programming techniques with multi-core processors, the cache hit rate jumped dramatically and drove non-linear performance improvements. Please visit http://www.intel.com/technology/advanced_comm/311566.htm for more details.

Measuring Code Performance

The code has been parallelized and now it is time to measure code performance using five code metrics: overall performance, cache hit rate, CPU utilization, synchronization overhead, and thread stall overhead.

Before examining the overall performance data, what should we reasonably expect? Given we migrated AMIDE from a single-core to a four-core system, the highest "realistic" expectation is a linear four-fold performance improvement. This is more-or-less a theoretical maximum because the cores share resources such as buses and memory which introduces execution delay. There is also overhead associated with maintaining data coherency among the caches in the system. Applying a rule-of-thumb, based on previous experience, these factors can lead to a 10-20% reduction in system performance. Therefore, on average, we anticipate a four-core system will perform in the range of 3.2 to 3.6 times faster than a single-core system.

Our first code metric is overall application performance, relative to the original code, as measured by the time it took VolPack to render images along the Z- and X-axes, shown in Figure 5. These two rotations were rendered from the same image dataset along two different axes. The outcome of our migration from a single core system to a four core system was a performance speedup of between 3.2 and 3.4 times. These results indicate this parallelization implementation was effective and reasonably well optimized.

The second code metric is a review of the L2 cache hit rate. The L2 cache hit rate was measured using the Intel VTune Performance Analyzer tool.

Figure 6 shows the L2 cache hit rate for the images rendered along the Z- and X-axes run on one core, two cores and four cores. Along the Z-axis, the cache hit rate is fairly consistent for one, two and four cores at about 76%. Along the X-axis, the L2 cache rate dips down to 34% with four cores. This lower L2 cache hit may be further evidence of the slower rendering time for X-axis renders shown in Figure 5.

The third code metric is CPU utilization. The CPU utilization was measured using the TOP command in Linux. During image rendering, all cores are running at nearly 100% utilization for each configuration: one-thread, two-thread and four-thread. All available cores are sharing evenly in the workload.

Our fourth code metric is synchronization overhead. This provides a measure of the non-compute resources required to maintain the threads which otherwise could be used to speed up the application. Figure 7 (on the next page) showing Thread 1 loading the image for 45 seconds and then spawning threads to cores 2, 3 and 4, was generated by the Intel® Thread Profiler tool. The solid green areas represent individual image renders where the renders start around the 46, 48, 56, 59, 62 and 68 second marks.

	One Core (seconds)	Two Cores (seconds)	Four Cores (seconds)	Speedup Ratio: One Core/ Four Cores
Z-axis rendering time	1.85	0.96	0.55	3.4
X-axis rendering time	5.48	3.18	1.71	3.2

Figure 5. Rendering Performance

	One Core	Two Core	Four Cores
Z-axis render	78%	78%	76%
X-axis render	26%	28%	34%

Figure 6. L2 Cache Hit Rates

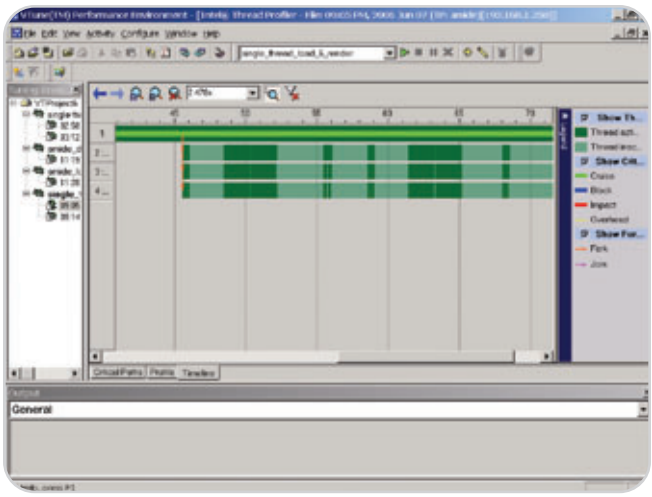


Figure 7. Synchronization Overhead: Parallel Image Rendering

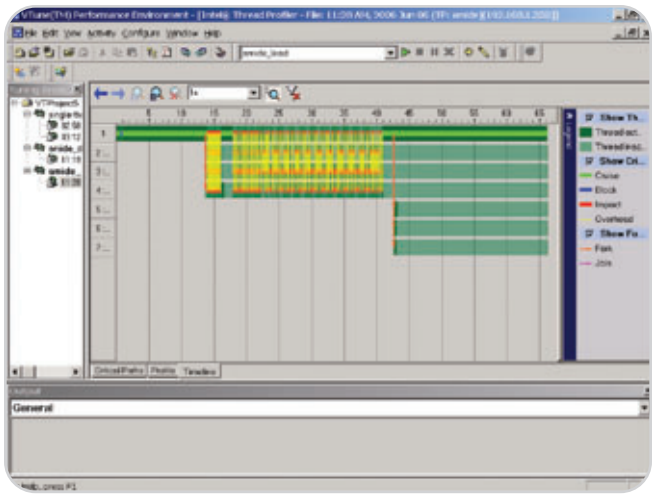


Figure 8. Synchronization Overhead: Parallel Image Loading and Rendering

Synchronization overhead is displayed in red when its duration is greater than 30 microseconds. The only synchronization overhead instance displayed in Figure 7 occurs when Thread 1 creates the threads for Core 2, 3 and 4 around time the 46 second time mark. We were pleased with this relatively low level of synchronization overhead.

In order to illustrate what a high synchronization overhead application would look like, we show an initial parallelization we performed on image loading within AMIDE. Each iteration of the loop (i.e., allocating and writing to data) required locking, introducing substantial synchronization overhead. The first time period displayed in Figure 8 shows the substantial synchronization overhead (in red) when the image loading was parallelized; the second time period is the parallelized image rendering similar to Figure 7. This is an unacceptable level of synchronization overhead occurring about 90% of the time during times 14 through 41 as shown by the high concentration of red lines.

Given a choice to optimize image loading which occurs once, and image rendering which executes many times as the image is viewed from different angles, we concentrated our efforts on rendering.

Our last code metric is thread stall overhead which provides a measure of the core idle time due to the threads waiting for system resources or work assignments. Figure 9 shows that a single thread CL.1, executes for 66 seconds; this is the sum of the time Thread 1 loads the image and the idle time between rendering images. For the vast majority of the rest of the time, four threads are executing, (CL.4) corresponding to image rendering. This means we have four cores all working away during image rendering and this is good.

There are short periods when only two or three threads CL.2 and CL.3, are working, but these are relatively short. CL.2 and CL.3 periods are better than CL.1 periods (one thread), but not as good as CL.4 (four threads). Thread profiling tools typically provide more detailed views of thread stalls, and they can map thread stalls to the responsible source code.

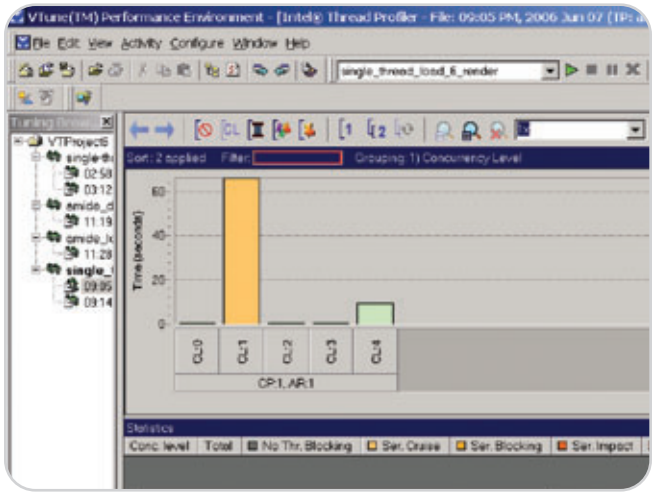


Figure 9. Thread Stall Overhead

Conclusion

The development of parallel code for multi-core processors warrants a careful consideration of threading approaches and a thorough analysis of the resulting performance. With multiple threads executing simultaneously, there are many opportunities for bottlenecks and resource contention issues. Code analysis tools can help identify these problem areas more quickly.

The principles described in this paper can be applied to any application transition from serial to multithreaded code. Embedded applications such as print imaging, virus detection, and call processing are good examples of applications that lend themselves to multithreading practices. These applications can benefit from creating multiple threads that run on multiple cores to process many requests simultaneously.

This code migration study showed it is possible to take an application designed to run on a single-core processor and migrate it to a four-core system in a matter of days. The performance increased by more than three times. This result was accomplished by using an efficient approach to parallelize and optimize the code and by employing tools suited to debugging and profiling threaded code.

Appendix 1: Software Analysis Tools

Sophisticated tools are available to provide the code metrics previously discussed. The Intel® Thread Profiler, Intel® Thread Checker and Intel® VTune™ Performance Analyzer are available to assist software developers.

The Intel Thread Checker checks for storage conflicts and looks for places where threads may lock or stall.

- Helps quickly identify shared and private variable conflicts.
- Isolates threading bugs to the source code line where the bug occurs.
- Describes possible causes of threading errors and suggested solutions with one-click diagnostic help.

The Intel Thread Profiler collects data on individual threads to help programmers check for load imbalances, lock contentions, synchronization bottlenecks, and parallelization overhead.

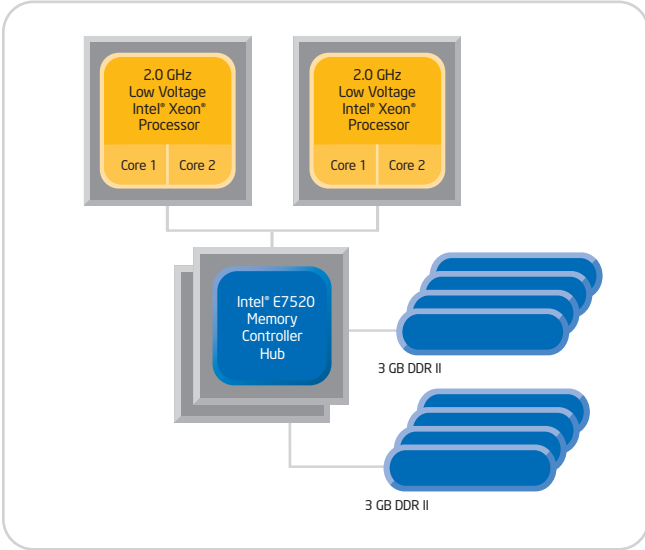
- Shows the application's critical path as it moves from thread to thread, helping the developer decide how to use threads more efficiently.
- Identifies synchronization issues and excessive blocking time that cause delays for Win32*, POSIX* threaded and OpenMP* code.
- Shows thread workload imbalances.

The Intel VTune Performance Analyzer measures performance at the CPU core and thread level.

- Use sampling to gain an accurate representation of the software's actual performance, with negligible overhead. No special builds or instrumentation are required.
- View results of time and event sampling on multiple levels to identify specific bottlenecks. Drill down to the exact operating system process, thread, module executable, or individual line of source code.
- Balance the workload after the thread view shows how much CPU is used by each thread and CPU idle time.
- Measure thread overhead and thread synchronization.

For more detailed information about these tools, threading practices and software programming guides, please visit http://www3.intel.com/cd/software/products/asmo-na/eng/index.htm

Appendix 2: Hardware Configuration



Platform Configuration:

2 x Dual-Core Intel® Xeon® Processor LV 2.0 GHz, Intel® E7520 MCH, DDR2 - 400 MHz 3072M, 8 DIMMS, each with 512 MB memory, 667 MHz Front-Side Bus Speed, Red Hat* Linux* Enterprise Linux 4.0 Update 1 (kernel: 2.6.9-11ELsmp), Intel® VTune™ Performance Analyzer 8.0 (Remote Data Collector), GNU* GCC version 3.4.3 BIOS: ALGHB025.

[1] We would like to extend our appreciation to Seven Pinnacles developers Diana Franklin and John Seng, who also serve as professors at The California Polytechnic State University, San Luis Obispo, for their contributions to this parallelization activity.

[2] Our changes to the VolPack code can be downloaded from http://www.sevenpinnacles.com/threaded_volpack. The images we benchmarked can be found at http://prdownloads.sourceforge.net/amide/m2862-small.tgz

[3] For more information, please visit http://graphics.stanford.edu/software/volpack.

[+] Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel Performance Benchmark Limitations (http://www.intel.com/performance/resources/limits.htm).

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications. Intel may make changes to specification and product descriptions at any time, without notice.

Information regarding third party products is provided solely for educational purposes. Intel is not responsible for the performance or support of third party products and does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices or products.

Copyright © 2007 Intel Corporation. All rights reserved.

Intel, the Intel logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel Xeon, and Intel VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Printed in USA 0107/LS/OCG/XX/PDF ♻ Please Recycle 315697-001US