Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel® OpenMP Compiler

Xinmin Tian[1], Yen-Kuang Chen[2,3], Milind Girkar[1], Steven Ge[3], Rainer Lienhart[2], Sanjiv Shah[4]

[1]Intel Compiler Labs, Software Solution Group, Intel Corporation
[2]Microprocessor Research, Intel Labs, Intel Corporation
[1,2]3600 Juliette Lane, Santa Clara, CA 95052, USA
[3]Intel China Research Center, Intel Corporation
[4]KAI Software Lab, Intel Corporation, 1906 Fox Drive, Champaign, IL 61820, USA
{Xinmin.Tian, Yen-kuang.Chen, Milind.Girkar, Steven.Ge, Rainer.Lienhart, Sanjiv.Shah}@intel.com

Abstract

Processors with Hyper-Threading technology can improve the performance of applications by permitting a single processor to process data as if it were two processors by executing instructions from different threads in parallel rather than serially. However, the potential performance improvement can be only obtained if an application is multithreaded by parallelization techniques. This paper presents the threaded code generation and optimization techniques in the Intel® C++/Fortran compiler. We conduct the performance study of two multimedia applications parallelized with OpenMP pragmas and compiled with the Intel compiler on the Hyper-Threading technology (HT) enabled Intel single-processor and multi-processor systems. Our performance results show that the multithreaded code generated by the Intel compiler achieved up to 1.28x speedups on a HT-enabled single-CPU system and up to 2.23x speedup on a HT-enabled dual-CPU system. By measuring IPC (Instructions Per Cycle), UPC (Uops Per Cycle) and cache misses of both serial and multithreaded execution of each multimedia application, we conclude three key observations: (a) the multithreaded code generated by the Intel compiler yields a good performance gain with the parallelization guided by OpenMP pragmas or directives; (b) exploiting thread-level parallelism (TLP) causes inter-thread interference in caches, and places greater demands on memory system. However, with the Hyper-Threading technology hides the additional latency, so that there is a small impact on the whole program performance; (c) Hyper-Threading technology is effective on exploiting both task- and data-parallelism inherent in multimedia applications.

1. Introduction

Modern processors become faster and faster, processor resources, however, are often underutilized by many applications and the growing gap between processor frequency and memory speed causes memory latency to become an increasing challenge of the performance. Simultaneous Multi-Threading (SMT) [7, 15] was proposed to allow multiple threads to compete for and share all processor's resources such as caches, execution units, control logic, buses and memory systems. The Hyper-Threading technology (HT) [4] brings the SMT idea to the Intel architectures and makes a single physical processor appear as two logical processors with duplicated architecture state, but with shared physical execution resources. This allows two threads from a single application or two separate applications to execute in parallel, increasing processor utilization and reducing the impact of memory latency by overlapping the latency of one thread with the execution of another

Hyper-Threading technology-enabled processors offer significant performance improvements for applications with a high degree of thread-level parallelism without sacrificing compatibility with the existing software or single-threaded performance. These potential performance gains are only obtained, however, if an application is efficiently multithreaded. The Intel® C++/Fortran compilers support OpenMP* directive- and pragma-guided parallelization, which significantly increase the domain of various applications amenable to effective parallelism. A typical example is that users can use OpenMP parallel sections to develop an application where *section-A* calls an integer-intensive routine and where *section-B* calls a floating-point intensive routine, so the performance improvement is obtained by scheduling *section-A* and *section-B* onto two different logical processors that share the same physical processor to fully utilize processor resources with the Hyper-Threading technology. The OpenMP directives or pragmas have emerged as the de facto standard of expressing thread-level parallelism in applications as they substantially simplify the notoriously complex task of writing multithreaded applications. The OpenMP 2.0 standard API [6, 9] supports a multi-platform, shared-memory, parallel programming paradigm in C++/C and Fortran95 on all popular operating systems such as Windows NT, Linux, and Unix. This paper describes threaded code generation techniques for exploiting parallelism explicitly expressed by OpenMP pragmas/directives. To validate the effectiveness of our threaded code generation and optimization techniques, we also characterize and study two workloads of multimedia applications parallelized with OpenMP pragmas and compiled with the Intel OpenMP C++ compiler on Intel Hyper-Threading architecture. Two multimedia workloads, including Support Vector Machine (SVM) and Audio-Visual Speech Recognition (AVSR), are optimized for the Intel® Pentium® 4 processor. One of our goals is to better explain the performance gains that are possible in the media applications through exploring the use of Hyper-Threading technology with the Intel compiler.

The remainder of this paper is organized as follows. We first give a high-level overview of Hyper-Threading technology. We then

®Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

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

present threaded code generation and optimization techniques developed in the Intel C++ and Fortran product compilers for the OpenMP pragma/directive guided parallelization, which includes the *Multi-Entry Threading* (*MET*) technique, lifting read-only-memory-references optimization for minimizing the data-sharing overhead among threads, exploitation of nested parallelism, and workqueuing model extension for exploiting irregular-parallelism. Starting from Section 4, we characterize and study two workloads of multimedia applications parallelized with OpenMP pragmas and compiled with the Intel OpenMP C++ compiler on Hyper-Threading technology enabled Intel architectures. Finally, we show the performance results of two multimedia applications.

2. Hyper-Threading Technology

Hyper-Threading technology brings the concept of Simultaneous Multi-Threading (SMT) to Intel Architecture. Hyper-Threading technology makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors [4]. From a software or architecture perspective, this means operating systems and user programs can schedule threads to logical CPUs as they would on multiple physical CPUs. From a microarchitecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources [4].

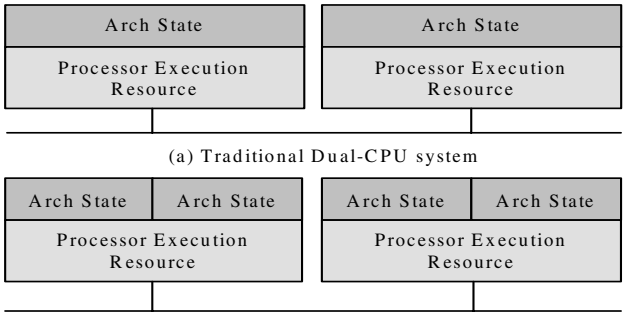

Figure 1: Traditional DP system vs. HT-capable DP system

The optimal performance is provided by the Intel NetBurst™ microarchitecture while executing a single instruction stream. A typical thread of code with a typical mix of instructions, however, utilizes only about 50 percent of execution resources. By adding the necessary logic and resources to the processor die in order to schedule and control two threads of code, Hyper-Threading technology makes these underutilized resources available to a second thread, offering increased system and application performance. Systems built with multiple Hyper-Threading enabled processors further improve the multiprocessor system performance, processing two threads for each processor.

Figure 1(a) shows a system with two physical processors that are not Hyper-Threading technology-capable. Figure 1(b) shows a system with two physical processors that are Hyper-Threading technology-capable. In Figure 1(b), with a duplicated copy of the architectural state on each physical processor, the system appears to have four logical processors. Each logical processor contains a complete set of the architecture state. The architecture state consists of registers including the general-purpose register group, the control registers, advanced programmable interrupt controller (APIC) registers, and some machine state registers. From a software perspective, once the architecture state is duplicated, the processor appears to be two processors. The number of transistors required to store the architecture state is a very small fraction of the total. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses. Each logical processor has its own interrupt controller or APIC. Interrupts sent to a specific logical processor are handled only by that logical processor.

With the Hyper-Threading technology, the majority of execution resources are shared by two architecture states (or two logical processors). Rapid execution engine process instructions from both threads simultaneously. The Fetch and Deliver engine and Reorder and Retire block partition some of the resources to alternate between the two intra-threads. In short, the Hyper-Threading technology improves performance of multi-threaded programs by increasing the processor utilization of the on-chip resources available in the Intel NetBurst™ microarchitecture.

3. Parallelizing Compiler

The Intel compiler incorporates many well-known and advanced optimization techniques [14] that are designed and extended to fully leverage Intel processor features for higher performance. The Intel compiler has a common intermediate representation (named IL0) for C++/C and Fortran95 language, so that the OpenMP directive- and pragma-guided parallelization and a majority of optimization techniques are applicable through a single high-level intermediate code generation and transformation, irrespective of the source language. In this Section, we present several threaded code generation and optimization techniques in the Intel compiler.

3.1 Threaded Code Generation Technique

We proposed and implemented a new compiler technology named *Multi-Entry Threading* (MET) [3]. The rationale behind MET is that the compiler does not create a separate compilation unit (or routine) for a parallel region/loop. Instead, the compiler generates a threaded entry (*T-entry*) and a threaded return (*T-return*) for a given parallel region and loop. We introduced three new graph nodes in the region-based graph, built on top of the control-flow graph. A description of these graph nodes is given as follows:

- *T-entry* denotes the entry point of a threaded code region and has a list of firstprivate, lastprivate, shared and reduction variables for sharing data among the threads.
- *T-ret* denotes the exit point of a threaded code region and guides the lower-level target machine code generator to adjust stack offset properly and give the control to the caller inside the multithreaded runtime library.
- *T-region* represents a threaded code region that is embedded in the original user routine.

The main motivation of the MET compilation model is to keep all newly generated multithreaded codes, which are captured by *T-entry*, *T-region* and *T-ret* nodes, embedded inside the user-routine without splitting them into independent subroutines. This method is different from *outlining* [10, 13] technique, and it provides later more optimization opportunities for higher performance. From the compiler-engineering point of view, the *MET* technique greatly reduces the complexity of generating separate routines in the Intel compiler. In addition, the *MET* technique minimizes the impact of OpenMP parallelizer on all well-known optimizations in the Intel compiler such as constant propagation, vectorization [8], PRE [12], scalar replacement, loop transformation, profile-feedback guided optimization and interprocedural optimization.

The code transformations and optimizations in the Intel compiler can be classified into (i) code restructuring and interprocedural optimizations (IPO); (ii) OpenMP directive-guided and automatic parallelization and vectorization; (iii) high-level optimizations (HLO) and scalar optimizations including memory optimizations such as loop control and data transformations, partial redundancy elimination (PRE), and partial dead store elimination (PDSE); and (iv) low-level machine code generation and optimizations such as register allocation and instruction scheduling. In Figure 2, we show a sample program using the *parallel sections* pragma.

```
void  parfoo( )
{   int  m, y, x[5000];   float   w, z[3000];
#pragma omp parallel sections shared(w, z, y, x)
   {     w =  floatpoint_foo(z, 3000);
     #pragma omp section
         y =  myinteger_goo(x, 5000) ;
   }
}
```

Figure 2. An Example with Parallel Sections

```
R-entry void  parfoo( )
{    … …
    __kmpc_fork_call(loc, 4, T-entry(__parfoo_psection_0), &w, z, x, &y)
    goto L1:
    T-entry void __parfoo_psection_0(loc, tid, *w, z[], *y, x[]) {
       lower = 0;   upper = 1; stride = 1;
       __kmpc_dispatch_init(…, tid, lower, upper, stride, ...);
     L33:
       t3 = __kmpc_dispatch_next(..., tid, &lower, &upper, &stride)
       if ((t3 & upper>=lower) != 0(SI32)) {
              pid = lower;
         L17:  if (pid == 0) {
                    *w = floatpoint_foo(z, 3000);
                } else if (pid == 1) {
                    *y = myinteger_goo(x, 5000);
                }
                pid = pid +1;
                __kmpc_dispatch_fini(…);
                if (upper >= pid)  goto L17
         goto L33
       }
       T-return;
    }
L1:    R-return;
}
```

Figure 3. Pesudo-Code After Parallelization

Essentially, the multithreaded code generator inserts the thread invocation call *__kmpc_fork_call*(…) with *T-entry* node and data environment (source line information *loc*, thread number *tid*, etc.) for each parallel loop, parallel sections or parallel region, and transforms a serial loop, sections, or region to a multithreaded loop, sections, or region, respectively. In this example, the *pre-pass* first converts *parallel sections* to a *parallel loop*. Then, the multithreaded code generator localizes loop lower-bound and upper-bound, privatizes the section *id* variable for the *T-region* marked with [*T_entry*, *T-ret*] nodes. For the *parallel sections* in the routine "*parfoo*", the multithreaded code generation involves (a) generating a runtime dispatch and initialization routine (*__kmpc_dispatch_init*) call to pass necessary information to the runtime system; (b) generating an enclosing loop to dispatch *loop-chunk* at runtime through the *__kmpc_dispatch_next* routine in the library; (c) localizing the loop lower-bound, upper-bound, and privatizing the loop control variable '*id*' as shown in Figure 3. Given that the granularity of the sections could be dramatically different, the static or static-even scheduling type may not achieve a good load balance. We decided to use the *runtime* scheduling type for a parallel loop generated by the pre-pass of multithreaded code generation. Therefore, the decision regarding scheduling type is deferred until run-time, and an optimal balanced workload can be achieved based on the setting of the environment variable OMP_SCHEDULE supported in the OpenMP library at run-time.

In order to generate efficient threaded-code that gains a speed-up over optimized uniprocessor code, an effective optimization phase ordering had been designed in the Intel compiler to make sure that optimizations, such as, IPO inlining, code restructuring, Igoto optimizations, and constant propagation, which can be effectively enabled before parallelization, preserve legal OpenMP program semantics and necessary information for parallelization. It also ensures that all optimizations after the OpenMP parallelization, such as auto-vectorization, loop transformation, PRE, and PDSE, can effectively kick in to achieve a good cache locality and to minimize the number of redundant computations and references to memory. For example, given a double-nested OpenMP parallel loop, the parallelizer is able to generate multithreaded code for the outer loop, while maintaining the symbol table information, loop structure, and memory reference behavior for the innermost loop. This enables the subsequent auto-vectorization for the innermost loop to fully leverage the SIMD Streaming Extension (SSE and SSE2) features of Intel processors [3, 8]. There are many efficient threaded-code generation techniques that have been developed for OpenMP parallelization in the Intel compiler. The following sub-sections describe some such techniques.

3.2 Lifting Read-Only Memory References

In this Section, we present an optimization LRMR that lifts read-only memory de-references from inside of a loop to outside the loop. The basic idea is that we pre-load a memory de-reference to a register temporary right after T-entry, if the memory reference is read-only. See the OpenMP Fortran code example in Figure 4.

```
real allocatable:: x(:,:)
… …
!$omp parallel do shared(x), private(m,n)
do  m=1, 100                !!  Front-End creates a dope-vector for allocatable
   do n=1, 100              !!  array x
      x(m, n) = …     ➔  dv_baseaddr[m][ n] = …
   end do
end do
… …
T-entry(dv_ptr …)   !! Threaded region after multithreaded code generation
    … …
    t1 = (P32 *)dv_ptr->lower                    !! dv_ptr is a pointer that points
    t2 = (P32 *)dv_ptr->extent                   !! dope-vector of array x
    do prv_m=lower, upper
       do prv_n =1, 100                          !! EXPR_lower(x(m,n))  = t1
          (P32 *)dv_ptr[prv_m][prv_ n] = …       !! EXPR_stride(x(m,n))  = t2
       end do
   end do
T-return
```

Figure 4. Example of Lifting Read-Only Memory References

The benefit of this optimization is that it reduces the overhead of a memory de-referencing, since the value is preserved in a register temporary for the read operation. In addition, another benefit is that it enables more advanced optimizations such if the memory de-references in array subscript expressions are lifted outside the loop. In Figure 4, for example, the address computation of array involves the memory de-references of the member *lower* and *extent* of the dope-vector, the compiler lifts the memory de-references of *lower* and *stride* outside the *m*-loop by analyzing and identifying the *read-only* memory references inside a parallel

region, *sections* or *do* loop. This optimization enables a number of optimizations such as software pipelining, loop unroll-and-jam, loop tiling, and vectorization, which results a good performance improvement in real large applications.

3.3 Static and Dynamic Nested Parallelism

Both static and dynamic nested parallelisms are supported by the OpenMP standard. However, most existing OpenMP compilers do not fully support nested parallelism, since the OpenMP-compliant implementation is allowed to serialize the nested inner regions, even when the nested parallelism is enabled by the environment variable OMP_NESTED or routine omp_set_nested(). For example, SGI compiler supports nested parallelism only if the loops are perfectly nested. PGI compiler does serialize the inner parallel regions. Given that broad classes of applications, such as imaging processing and audio/video encoding and decoding algorithms, have shown performance gains by exploiting nested parallelisms. We provided the compiler and runtime library support to exploit static and dynamic nested parallelism. Figure 5(a) shows a sample code with nested *parallel* regions, and Figure 5(b) does show the pseudo-threaded-code generated by the Intel compiler.

```
(a) A Nested Parallel Region Example
void nestedpar()
{   static double a[1000];   int    id;
#pragma omp parallel private(id)
    {       id = omp_get_thread_num();
#pragma omp parallel
            do_work(a, id, id*100);
    }
}
(b) Pseudo Multithreaded Code Generated by Parallelizer
entry extern void _nestedpar()
{ .....
  ___kmpc_fork_call(___nestedpar_par_region0)(P32));
  goto L30
    T-entry void __nestedpar_par_region0()
    {    t0 = _omp_get_thread_num();
         prv_id = t0;
        ___kmpc_fork_call(__nestedpar_par_region1)(P32), &prv_id)
        goto L20;
          T-entry void __nestedpar_par_region1(id_p)
          {    t1 = _do_work( &a, *id_p, *id_p * 100)
               T-return
          }
        L20:  T-return
    }
  L30:
    return
}
```

Figure 5. An Example of Nested Parallel Regions

As shown in Figure 5(b), there are two threaded regions, or T-regions, created within the original function nestedpar(). *T-entry __nestedpar_par_region0*() corresponds to the semantics of the outer *parallel* region, and the T-entry *__nestedpar_par_region1*() corresponds to the semantics of the inner *parallel* region. For the inner *parallel* region in the routine nested*par*, the variable *id* is a *shared* stack variable for the inner *parallel* region. Therefore, it is accessed and shared by all threads through the *T-entry* argument *id_p*. Note that the variable *id* is a private variable for the outer *parallel* region, since it is a local defined stack variable.

As we see in Figure 5(b), there are no extra arguments on the *T-entry* for sharing local static array '*a*', and there is no pointer de-referencing inside the *T-region* for sharing the local static array '*a*' among all threads in the teams of both the outer and inner *parallel* regions. This uses the optimization technique presented in [3] for sharing local static data among threads; it is an efficient way to avoid the overhead of argument passing across *T-entries*.

3.4 Exploiting Irregular Parallelism

Irregular parallelism inherent in many applications is hard to be exploited efficiently. The workqueuing model [1] provides a simple approach for allowing users to exploit irregular parallelism effectively. This model allows a programmer to parallelize control structures that are beyond the scope of those supported by the OpenMP model, while still fitting into the framework defined by the OpenMP specification. In particular, the workqueuing model is a flexible programming model for specifying units of work that are not pre-computed at the start of the worksharing construct. See a simple example in Figure 6.

```
void wq_foot(LIST *p)
{
#pragma intel omp parallel  taskq shared(p)
  {  while (p!= NUL:L)  {
        #pragma intel omp task captureprivate(p)
         {   wq_work1(p, 10);  }
        #pragma intel omp task captureprivate(p)
         {   wq_work2(p, 20);  }
         p= p->next;
     }
  }
}
```

Figure 6. A While-Loop with Workqueuing Pragmas

The *parallel taskq* pragma specifies an environment for the '*while loop'* in which to enqueue the units of work specified by the enclosed *task* pragma. Thus, the loop's control structure and the enqueuing are executed by single thread, while the other threads in the team participate in dequeuing the work from the *taskq* queue and executing it. The *captureprivate* clause ensures that a private copy of the link pointer *p* is captured at the time each task is being enqueued, hence preserving the sequential semantics. The workqueuing execution model is shown in Figure 7.

Essentially, given a program with workqueuing constructs, a team of threads is created, when a parallel region is encountered. With the workqueuing execution model, from among all threads that encounter a *taskq* pragma, one thread (T_K) is chosen to execute it initially. All the other threads (T_m, where m=1, …, N and m≠K) wait for work to be enqueued on the work queue. Conceptually, the *taskq* pragma causes an empty queue to be created by the chosen thread T_K, enqueues each ***task*** it encounters, and then the code inside the *taskq* block is executed single-threaded by the T_K. The *task* pragma specifies a unit of work, potentially executed by a different thread. When a task *pragma* is encountered lexically within a *taskq* block, the code inside the *task* block is enqueued on the queue associated with the *taskq*. The conceptual queue is disbanded when all work enqueued on it finishes, and when the end of the *taskq* block is reached.

The Intel C++ OpenMP compiler has been extended throughout its various components to support the workqueuing model for generating multithreaded codes corresponding to the workqueuing constructs as the Intel OpenMP extension. More code generation details for the workqueuing constructs are presented in the paper [1]. In the next Section, we describe the multimedia application SVM and AVSR modified with OpenMP pragmas for evaluating our multithreaded code generation and optimizations developed in the Intel compiler together with the Intel OpenMP runtime library.

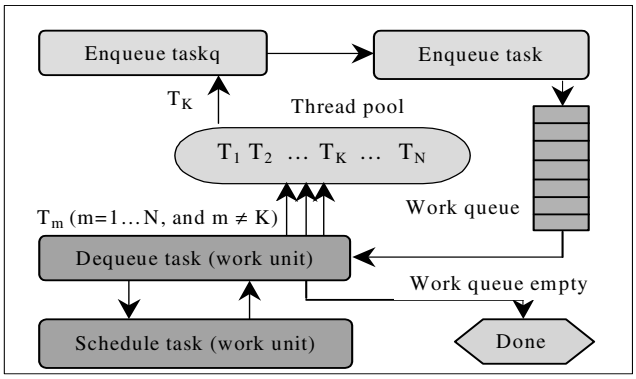

Figure 7. Workqueuing Execution Model

4. Multimedia Workloads

Due to the inherently sequential constitution of the algorithms of multimedia applications, most of the modules in these optimized applications cannot fully utilize all the execution units available in the off-the-shelf microprocessors. Some modules are memory-bounded, while some are computation-bounded. In this Section, we describe the selected multimedia workloads and discuss our approach of parallelizing the workloads with OpenMP.

4.1 Workload Description

4.1.1 Support Vector Machines

The first workload in our study is support vector machine (SVM) classification algorithm that is a well-known machine-learning algorithm [11]. Machine learning plays a key role in automatic content analysis of multimedia data. A common task is to predict the output y for an unseen input sample $\mathbf{x}$ given a training set $\{(x_i, y_i)\}_{i\in\{1,\ldots,N\}}$ consisting of input $\mathbf{x}_i \in \mathbf{R}^K$ and its desired output $y_i \in \{-1, +1\}$. The process of evaluating trained SVMs is like the following:

$$F(\mathbf{x}) = \mathrm{sign}\left[\left(\sum_{i=1}^{N} y_i \alpha_i \Phi(\mathbf{x}, \mathbf{x}_i)\right) + b\right]$$

where $\Phi(\mathbf{x}, \mathbf{x}_i)$ often is either linear $\Phi^L(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$ or radial basis function $\Phi^{RBF}(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / \sigma^2)$.

4.1.2 Audio-visual Speech Recognition

The second workload that we investigate is audio-visual speech recognition (AVSR). There are many applications using automatic speech recognition systems, from human computer interfaces to robotics. While computers are getting faster, speech recognition systems are not robust without special constraints. Often, robust speech recognition requires special conditions, such as, smaller vocabulary, or very clean signal of the voice.

In recent years, several speech recognition systems that use visual together with audio information showed significant increase in performance over the standard speech recognition systems. Figure 8 shows a flowchart of the AVSR process. The use of visual feature in AVSR is motivated by the bimodality of the speech formation and the ability of humans to better distinguish spoken sounds when both audio and video are available. Additionally, the visual information provides the system with complementary features that cannot be corrupted by the acoustic noise of the environment. In our performance study, the system developed by Liang *et al.* [2] is used.

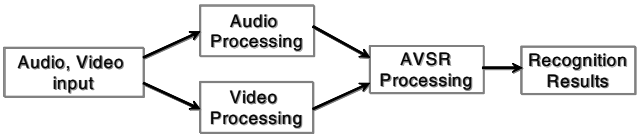

Figure 8. Process of Audio-Visual Speech Recognition

4.2 Data-Domain Decomposition

A way of exploiting parallelism of multimedia workloads is to decompose the work into threads in data-domain. As described in Section 4.1.1, the evaluation of trained SVMs is well-structured and can, thus, be multithreaded at multiple levels. On the lowest level, the dimensionality K of the input data can be very large. Typical values of K range between a few hundreds to several thousands. Thus, the vector multiplication in the linear, polynomial, and sigmoid kernels as well as the L_2 distance in the radial basis function kernel can be multithreaded. On the next level, the evaluation of each expression in the sum is independent of each other. Finally, in an application several samples are tested and each evaluation can be done in parallel. In Figure 9, we show the parallelized SVM by simply adding a *parallel for* pragma. The programmer intervention for parallelizing the SVM is minor. The compiler generates the multithreaded code automatically.

```
const int NUM_SUPP_VEC = 1000;   // Number of support vectors
const int NUM_VEC_DIM = 24*24;   // Feature vector size; 24x24 pixel window
// 1D signal scanned by sliding window for faces of size 24x24 pixels
const int SIGNAL_SIZE = 320*240;
const int NUM_SAMPLES = SIGNAL_SIZE-NUM_VEC_DIM+1;
Ipp32f supportVector[NUM_SUPP_VEC][NUM_VEC_DIM];
Ipp32f coeffs [NUM_SUPP_VEC];
Ipp32f samples[SIGNAL_SIZE];        // input signal array
Ipp32f result [NUM_SAMPLES];        // stores classification result
float linear_kernel(const Ipp32f* pSrc1, int len, int index)   // Linear   Kernel
{       Ipp32f tmp_result;
        ippsDotProd_32f(pSrc1, supportVector[index], len, &tmp_result);
        return tmp_result * coeffs[index];
}

void  main()
{       int blockSize = ...;
        for  (int jj=0 ; jj<NUM_SAMPLES; jj+=blockSize)  {
            for  (int i=0 ; i<NUM_SUPP_VEC; i+=1)  {
                  int loopEnd_j = std::_MIN(NUM_SAMPLES, jj+blockSize);
                  #pragma omp parallel for default(shared)
                  for  (int j=jj ; j<loopEnd_j ; j++)  {
                      result[j] += linear_kernel(&samples[j], NUM_VEC_DIM, i);
                  }
            }
        }
}
```

Figure 9. Exploiting Data-Parallelism of the SVM

4.3 Functional Decomposition

The functional decomposition is another way to multithread an application for exploiting task-parallelism. The AVSR application has clearly four different functional components. These are audio processing, video processing, audio-video processing, and others. Therefore, a natural scheme of parallelizing the AVSR is to map a functional component to an OpenMP *worksharing section* [6], as shown in Figure 10.

Streams of audio and video data can be broken into pieces and be processed in pipeline. In our multithreaded application, while the

audio processing and the video processing are working on the current piece of the data, the AVSR processing is working on the previous piece of the data as well. We did parallelize not only the parallel tasks, but also the pipeline tasks.

Same as exploiting data-parallelism in the SVM application, the programmer intervention for parallelizing the AVSR is also pretty small. A few OpenMP pragmas are simply added to the original source code. The compiler performs the threaded code generation presented in Section 3 together with the OpenMP library support to execute the AVSR application in parallel.

```
#pragma omp parallel sections default(shared)
{    #pragma omp section
       {    DispatchThreadProc( &AVSRThData );  }  // data input and dispatch
     #pragma omp section
       {   AudioThreadProc( &AudioThData );  }        // process audio data
     #pragma omp section
       {   VideoThreadProc( &VideoThData );    }      // process video data
     #pragma omp section
       {    AVSRThreadProc( &AVSRThData );  }       // do avsr
}
```

Figure 10. Exploiting Task-Parallelism of the AVSR

4.4 Exploiting Dynamic Nested Parallelism

In addition to functional-decomposition of the AVSR application, we exploit the nested data-parallelism in the dynamic extent of the video processing *section* (or *thread*). The major motivation of further partitioning this thread into multiple threads is to achieve better load balance. The execution time breakdown of the AVSR workload is shown in Figure 11 in which the video processing takes around half of the time. To exploit task-level parallelism of the application on a single processor with Hyper-Threading technology or a dual-processor system without Hyper-Threading technology, the workload can be balanced well by having the video processing thread on one processor and having the rest on the other processor. However, on a dual-processor system with Hyper-Threading technology, pure functional decomposition cannot have balanced loads. This is because video processing takes ~50% of the total execution time. We further make dot-product of matrices/vectors and Fourier transform into multiple threads, as shown in Figure 12. Thus, as shown in Figure 13, we have totally three threading schemes in our experiment to evaluate the exploitation of static nested parallelism supported by the Intel compiler and OpenMP runtime library.

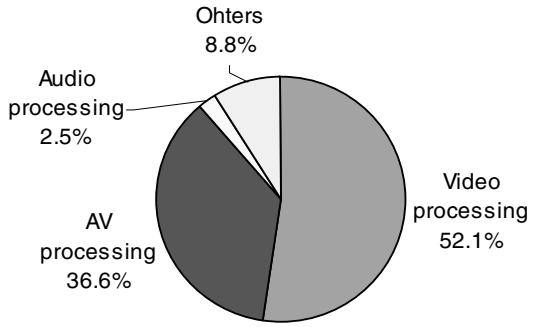

Figure 11. Execution time breakdown of the AVSR workload

Figure 13 shows the application AVSR parallelized with OpenMP pragmas to exploit task and data parallelisms, where, ***A*** stands for audio processing, ***V*** stands for video processing, ***AV*** stands for audio-video processing, and ***O*** stands for other miscellaneous processing. Figure 13(a) shows the multi-threading model when we only have four threads via functional decomposition. Figure 13(b) and (c) show the nested parallelism when video processing is further threaded into 2 or 4 threads. The bottom nodes denote the additional threads created for executing the parallel *for* loop within the dynamic extent of the *parallel sections*.

```
// In the parent function,  the dot-product kernel is called in a parallel sections
omp_set_nested( 1 );
:
call dot-product of matrix and vector kernel
:
// In the dot-product of matrix and vector
float          **matrix;     // input matrix
float          *vector,      // input vector
               *result;      // result vector
int            rows, columns;
// In this example the number of rows is 480, so we set chunk size to 120
// and use static scheduling for  each thread
#pragma omp parallel for num_threads(4) schedule(static, 120)
for (int i=0; i<rows; i++)
{
   ippmDotProduct_vv_32f(matrix[i], vector, &(result[i]), columns);
}
```

Figure 12. Exploiting Nested Parallelism of the AVSR

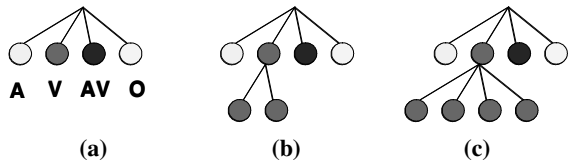

Figure 13: Task- and Data-Parallelism of the AVSR Workload

5. Performance

We conducted our performance evaluation with two multimedia applications to examine the performance of multithreaded codes generated by the Intel compiler. The generated codes are highly optimized with architecture-specific, advanced scalar and array optimizations assisted with aggressive memory disambiguation. Our results show that Hyper-Threading technology and the Intel compiler offer a cost-effective performance gain (10%~28%) for our applications on a single processor (SP+HT), and offer up to 2.23x speedup on a dual-processor system with Hyper-Threading technology-enabled (DP+HT). The performance measurement of two multimedia applications SVM and AVSR is carried out on a dual–processor HT-enabled Intel® Xeon™ system running at 2.0GHz, with 1024MB memory, an 8K L1-Cache, a 512K L2-Cache, and no L3-Cache. When we measure single-processor performance on a Dual-Processor (DP) system, we disable one physical processor from the BIOS. We disable the support of Hyper-Threading technology from the BIOS in order to measure the performance of our applications on the processor without using Hyper-Threading technology. To use the serial execution time as a base on the system experimentally in our lab setting, we disable one physical processor and Hyper-Threading technology, and run the highly optimized serial codes of applications.

Essentially, the performance scaling is derived from the serial execution (SP) with Hyper-Threading technology disabled and one physical processor disabled on our system. The multithreaded execution is done with three system configurations: (1) SP+HT (Single-Processor with HT-enabled), (2) DP (Dual Processor with HT-disabled), (3) DP+HT (Dual-Processor with HT-enabled). In Figure 14, we show the normalized speedup of our multithreaded execution of the SVMs (2 kernels). The workloads achieved very

Table 1: The workload characteristics of two multimedia applications on a SP or DP system with Hyper-Threading technology disabled , and a SP and DP system with Hyper-Threading technology enabled (SP+HT, DP+HT).

	SVM								AVSR	
	Linear				Radial Basis Function				SP	SP+HT (with 2-inner threads)
	SP	SP+HT	DP	DP+HT	SP	SP+HT	DP	DP+HT		
Clockticks (millions)	4,093	3,824	2,239	2,139	6,995	6,274	3,684	3,374	36,633	27,998
Instructions retired (millions)	3,152	3,174	3,202	3,594	4,337	4,392	4,384	4,487	19,415	19,599
IPC (Instructions Per Cycle)	0.77	0.83	1.43	1.68	0.62	0.7	1.19	1.33	0.53	0.70
UPC (Uops Per Cycle)	1.31	1.42	2.44	2.64	1.08	1.22	2.07	2.33	0.84	1.13
FP/MMX/SSE/SSE-2 (millions)	1,775	1,776	1,776	1,776	2,883	2,883	2,883	2,884	7,273	7,063
First-level cache load miss rate	2.7%	2.9%	3.0%	3.7%	3.0%	3.9%	3.2%	4.3%	11.0%	14.0%
2nd-level cache load miss rate	2.3%	4.1%	4.5%	5.3%	2.0%	3.1%	3.4%	3.8%	50.0%	26.6%

good performance gain using the Intel OpenMP C++ compiler for data-domain decomposition. For instance, from a single processor with HT-disabled to the single processor with HT-enabled, we achieve speedups ranging from 1.10x to 1.13x with 2-thread run. The speedup ranges from 1.92x to 1.97x for 2-thread run with DP configuration. The speedup ranges from 2.13 to 2.23x for 4-thread run with DP+HT configuration. This indicates that we utilize the microprocessor more efficiently.

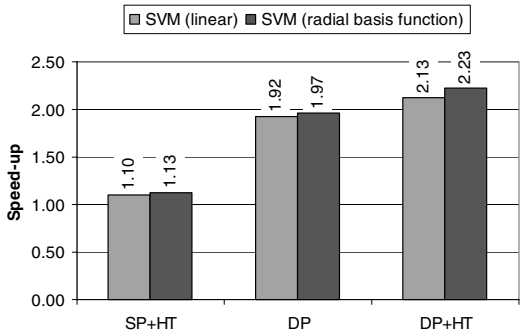

Figure 14. Speedup of Multithreaded SVMs

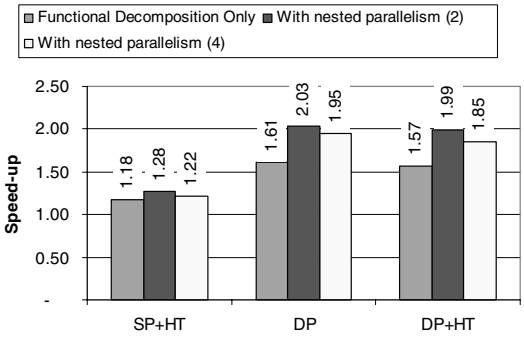

Figure 15. Speedup of the Multithreaded AVSR

Figure 15 shows the speedup of the OpenMP version of the AVSR with different amount of nested parallelism under different system configuration. Again, by changing from a single processor Hyper-Threading technology disabled to the single processor with Hyper-Threading technology-enabled, a speedup ranging from 1.18 to 1.28x is achieved with 2 threads under the SP+HT configuration. The speedup is 1.61x for 4 outer threads, 2.03x for 4 outer, 2 inner threads, and 1.95x for 4 outer, 4 inner threads with the DP configuration. The speedup is 1.57x for 4 outer threads, 1.99x for 4 outer, 2 inner threads, and 1.85x for 4 outer, 4 inner threads with DP+HT configuration. Clearly, we achieved ~2x speedup from a single-CPU system to a dual-CPU system.

One observation we have from Figure 15 is that the best speedup of AVSR workload with DP+HT configuration is 1.97% lower than the best speedup of the AVSR with the DP configuration. It attributes to one cause, that is, only three logical processors are effectively used when the *A* (2.5%) and *O* (8.8%) are completed for 4-outer-2-inner-thread execution. This means that the benefit from one physical processor with HT-enabled, which is evidenced with the performance gain under SP+HT configuration, is not enough to counteract the penalty of one idle logical processor caused by the unbalanced load. Our observation applies to the 4-outer-4-inter-thread execution scheme as well. The challenge here is how to exploit parallelism in *AV* (36.6%), which is one of our future research topics beyond the scope of this paper.

Another observation we have from Figure 15 is that the speedup from the 4 outer and 2 inner threads is better than the speedup from the 4 outer and 4 inner threads under both DP and DP+HT configurations. This is simply due to the less threading overheads are introduced with a smaller number of threads. Later, we discuss more about controlling parallelism and controlling spin-waiting for getting a good trade-off between benefits and costs. In any case, we have achieved ~2x speedup under both DP and DP+HT configurations.

Functional decomposition may not deliver the best performance due to unbalanced load of all tasks among all processors in the system. Given the inherent variation of granularity for each task (or module), it is hardly to achieve the best potential performance without exploiting another level of parallelism. Essentially, for media workloads, we can exploit data-parallelism to overcome the issue of exploiting task-parallelism. As we show in Figure 15, by exploiting the inner parallelism with data-domain decomposition, we achieve much better speedups -- the performance gain is around 40% with the 4 outer and 2 inner threads comparing to 4 outer threads (exploiting task-parallelism only). Thus, exploiting nested-parallelism is necessary to achieve better load balance and speedup. (Note: the inner-parallelism does not have to be data-parallelism always; it can be task-parallelism as well.) On the other hand, Figure 15 also shows that excessive threads introduce more extra threading overhead, the performance improvement with 4 inner threads is not better than that with 2 inner threads. Therefore, effectively controlling parallelism is still an important aspect to achieve the desired performance on a HT-enabled Intel Xeon processor system, even though the potential parallelism could improve the processor utilization. With Intel compiler and runtime, users are allowed to control how much time each thread should spend spinning at run-time. An environment variable KMP_BLOCKTIME is supported in the library. Also, the spinning time can be adjusted by using the kmp_set_blocktime() API call at

runtime. On a HT-enabled processor more than one thread can be executing on the same physical processor at the same time. This indicates that both threads have to share that processor's resources. It makes spin-waiting extremely expensive since the thread that is just waiting is now taking valuable processor resources away from the other thread that is doing useful work. Thus, when exploring the use of Hyper-Threading technology, the *block-time* should be very short so that the waiting thread sleeps as soon as possible allowing still useful threads to more fully utilize all processor resources. In our previous work, we use Win32 Threading Library calls to parallelize our multimedia workloads [5]. While we can achieve good performance, multi-threading them takes a huge amount of effort. With the Intel OpenMP compiler and OpenMP runtime library support, we demonstrated same or better performance with much less effort. In other words, the programmer intervention for parallelizing our multimedia applications is pretty minor.

Furthermore, we characterize the multimedia workloads by using Intel VTune Performance Analyzer under SP, SP+HT, DP, and DP+HT configurations to examine the HT benefits and costs instead of presenting speedup only. As shown in Table 1, although the numbers of instructions retired and cache miss rates (e.g., 2.7% vs 2.9% first-level cache miss rates for the linear SVM) are increased for both applications after threading due to execution resource sharing, cache and memory sharing, and contention, the overall application performance still increases. More specifically, the IPC is improved from 0.77 to 0.83 (8%) for SVM (linear) on SP, 17% for SVM (linear) on DP, 13% for SVM (RBF) on SP, 12% for SVM (RBF) on DP, and 30% for AVSR on SP. These results indicate the processor resource utilization is greatly improved for our multimedia applications with the Hyper-Threading technology.

6. Conclusions

In this paper, we presented a set of implemented compilation techniques that are unique to the Intel high-performance compiler for OpenMP pragma-guided and directive-guided parallelization. Two multimedia applications are studied to demonstrate and evident that the multithreaded codes generated and optimized by the Intel compiler are very efficient, together with the support of the well-tuned Intel OpenMP runtime library. The performance improvements achieved on three SP+HT, DP and DP+HT system configurations are very impressive for the multimedia applications (SVM and AVSR) studied in this paper. The performance results and workload characteristics of SVM and AVSR demonstrated and evidenced our three main observations: (a) the multithreaded code generated by the Intel compiler yields a good performance gain with the parallelization guided by the OpenMP pragmas; (b) the exploited thread-level parallelism (TLP) causes inter-thread interference in caches, and places greater demands on the memory system. However, the Hyper-Threading technology hides the additional latency, so that there is only a very small impact on the whole program performance, and the overall performance gain makes this little impact not visible on Hyper-Threading enabled Intel platforms; (c) Hyper-Threading technology is effective on exploiting both task- and data-parallelism through functional and data decomposition in multimedia applications.

Acknowledgments

The authors thank all members of the Intel compiler team for their great work in developing the Intel high-performance compiler. In particular, we thank A. Bik, P. Grey, E. Su, H. Saito, D. Schouten for their contribution in PAROPT projects, M. Domeika and D. King for the C++ FE support, B. Shankar and M. L. Ross for the Fortran FE support, K. J. Kirkegaard for IPO support, and Z. Ansari for PCG support. Special thanks go to P. Petersen, G. Habb and the library team at KSL for developing the OpenMP library. We would like to thank L. Liang, X. Liu, X. Pi, A. Nefan, and P. Liou for the development of speech recognition workloads.

References

[1] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen, "Compiler Support for Workqueuing Execution Model for Intel SMP Architectures", in *Proc. of European Workshop on OpenMP (EWOMP)*, Sep. 2002.

[2] L. Liang, X. Liu, M. Zhao, X. Pi, and A. V. Nefian, "Speaker Independent Audio-Visual Continuous Speech Recognition," in *Proc. of Int'l Conf. on Multimedia and Expo*, vol. 2, pp. 25-28, Aug. 2002.

[3] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance", *Intel Technology Journal,* Q1, 2002. (http://www.intel.com/technology/itj)

[4] D. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-Threading Technology Microarchitecture and Architecture," *Intel Technology Journal,* Vol. 6, Q1, 2002.

[5] Y.-K. Chen, M. Holliman, E. Debes, S. Zheltov, A. Knyazev, S. Bratanov, R. Belenov, and I. Santos, "Media Applications on Hyper-Threading technology," *Intel Technology Journal*, Q1 2002.

[6] OpenMP Architecture Review Board, "OpenMP C++ Application Program Interface," V2.0, Mar. 2002. (http://www.openmp.org)

[7] D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," in *Proc. of Micro-34,* Dec. 2001.

[8] A. Bik, M. Girkar, P. Grey, and X. Tian, "Automatic Intra-Register Vectorization for the Intel® Architecture," *Inter'l Journal of Parallel Programming*, vol. 30, no. 2, Apr. 2002.

[9] OpenMP Architecture Review Board, "OpenMP Fortran Application Program Interface," V2.0, Nov. 2000. (http://www.openmp.org)

[10] C. Brunschen and M. Brorsson, "OdinMP/CCp–A Portable Implementation of OpenMP for C," in *Proc. of European Workshop on OpenMP (EWOMP),* Sep. 1999.

[11] C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition", *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121-167, Jun. 1998.

[12] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu, "A new algorithm for partial redundancy elimination based on SSA form," in *Proc. of ACM Conf. on Programming Language Design and Implementation (SIGPLAN),* pp. 273-286, Jun. 1997.

[13] J.-H. Chow, L. E. Lyon, and V. Sarkar, "Automatic Parallelization for Symmetric Shared-Memory Multiprocessors, in *Proc. of CASCON*, pp. 76-89, Nov. 1996.

[14] M. J. Wolfe, *High Performance Compilers for Parallel Computers,* Addison-Wesley Publishing Company, Redwood City, CA, 1996.

[15] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", In Proc. of *Int'l Symp. on Computer Architecture*, pp. 392-403, Jun. 1995.