Volume 11 Issue 03 Published, August 22, 2007 ISSN 1535-864X DOI: 10.1535/itj.1103.02

Tera-scale Computing

Accelerator Exoskeleton

Accelerator Exoskeleton

Perry Wang, Corporate Technology Group, Intel Corporation
Jamison Collins, Corporate Technology Group, Intel Corporation
Gautham Chinya, Corporate Technology Group, Intel Corporation
Hong Jiang, Mobility Group, Intel Corporation
Xinmin Tian, Software Solutions Group, Intel Corporation
Milind Girkar, Software Solutions Group, Intel Corporation
Lisa Pearce, Mobility Group, Intel Corporation
Guei-Yuan Lueh, Mobility Group, Intel Corporation
Sergey Yakoushkin, Corporate Technology Group, Intel Corporation
Hong Wang, Corporate Technology Group, Intel Corporation

ABSTRACT

To maximize performance and power efficiency, future multi-core architectures may be heterogeneous, incorporating some accelerator cores alongside the IA cores. Accelerator Exoskeletons provide a shared virtual memory heterogeneous multi-threaded programming paradigm for these accelerators using novel CPU instruction set extensions and software tool chains with an Intel® Architecture (IA) look-n-feel. Firstly, we introduce the proposed architectural extensions known as the *Exoskeleton Sequencer* (EXO), which represents heterogeneous accelerators as ISA-based MIMD architecture resources, and a shared virtual memory heterogeneous multi-threaded program execution model that tightly couples specialized accelerator cores with general-purpose CPU cores. Then we introduce the *C for Heterogeneous Integration* (CHI) programming environment that includes a compiler, runtime, debugger, and performance-analysis tools. The CHI compiler extends the OpenMP pragma for heterogeneous multi-threading programming, and it produces a single fat binary with code sections corresponding to different instruction sets. The runtime can judiciously spread parallel computation across the heterogeneous cores to optimize performance and power.

INTRODUCTION

The relentless pace of Moore's Law will lead to mainstream multi-core microprocessor designs with extensive on-die integration of a large number of cores [11]. Fundamentally, to scale multi-core processor designs to incorporate a large number of cores, ultra low Energy Per Instruction (EPI) cores are essential [6]. One approach to improving EPI by an order of magnitude is through heterogeneous multi-core design, in which some cores vary in functionality, instruction set (ISA), performance, power, and energy efficiency [14]. The key challenge then becomes how to accomplish such heterogeneous integration and achieve high performance while still maintaining the look-n-feel of the classic mainstream IA-based programming models and software ecosystem.

In this paper we present an overview of EXOCHI: *Exoskeleton Sequencer* (EXO), an architecture proposal to represent heterogeneous accelerators as ISA-based MIMD architectural resources, and *C for Heterogeneous Integration* (CHI), a programming environment that supports tightly coupled integration of heterogeneous cores. The EXO architecture supports the familiar POSIX shared virtual memory multi-threaded programming model for heterogeneous cores. Architecturally, the heterogeneous cores are exposed to the programmer as a new form of sequencer resource. They can be regarded as application-level MIMD functional units on which user-level threads, or *shreds*, encoded in the accelerator-specific ISA can execute. Having a shared virtual address space between the IA sequencer and accelerator sequencers facilitates code and data sharing and harmonizes cooperation between the concurrent shreds of different ISAs. Such a program is said to be *multi-shredded*.

The CHI integrated programming environment allows an application developer to inline blocks of accelerator-specific assembly or domain-specific language with traditional C/C++ code. The CHI compiler produces a single fat binary consisting of executable code sections corresponding to the different ISAs. CHI further extends the OpenMP pragmas [21, 23, 26] to allow the programmer to express thread-level parallelism by demarcating parallel regions of code targeting

heterogeneous accelerators. The CHI extensions to OpenMP support both fork-join and producer-consumer parallelism among the accelerator shreds and between the IA shreds and the accelerator shreds. The CHI runtime can judiciously spread the shreds across the heterogeneous sequencers dynamically to maximize throughput performance while minimizing power.

The rest of the paper is organized as follows. We first briefly review related work. We then introduce the EXO architecture that supports a shared virtual memory heterogeneous multi-threaded programming model. We then present an overview of the CHI integrated programming environment that extends the Intel® C++ Compiler, runtime, and tool chains to provide the familiar IA look-n-feel to program heterogeneous cores. To prototype the EXO architecture, we describe potential heterogeneous multi-core processors which combine an Intel® Core™2 Duo processor [27] and two possible accelerators: an 8-core 32-thread Intel® Graphics Media Accelerator (GMA) X3000 [10] or the Datastream Processing Engine (DPE) from a research Scalable Communication Core (SCC) prototype [8]. We demonstrate code examples and evaluate performance.

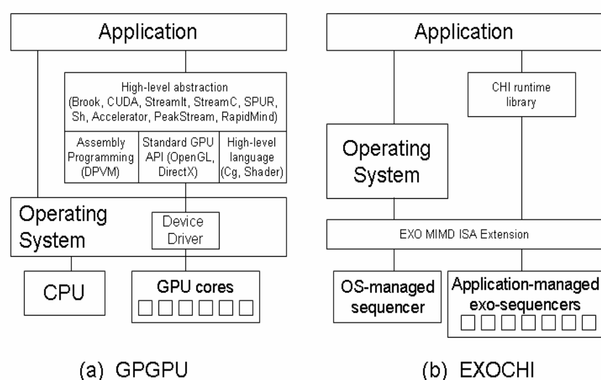

Figure 1: Alternate programming environments

RELATED WORK

There has been a rich body of research on heterogeneous acceleration. In most published work, the execution models usually fall into two classifications: (category 1) an ISA-based tightly coupled approach or (category 2), a device driver-based loosely coupled execution model. An example of the tightly coupled approach is the Software-configurable Processor (SCP) architecture [4] in which a custom ISA extension represents the operations implemented by a hardware accelerator attached to the CPU. The CPU is then responsible for sequencing, decoding, and dispatching each co-processor instruction, stalling until the co-processor execution completes. This approach resembles the classic x87 *escape-wait* style co-processor instruction execution where the co-processor does not sequence instructions independently from the CPU.

Examples of the second category include most known GPGPU infrastructures [1, 3, 5, 13, 15, 16, 17, 18, 19, 20, 22, 24, 25, 28]. As depicted in Figure 1(a), the CPU resources (cores and memory) are managed by the operating system (OS), and the GPU resources are separately managed by vender-supplied device drivers. Applications and device drivers run in separate address spaces, and consequently, data communication and synchronization between them is usually carried out in coarse granularity through explicit data copying via device driver APIs. In the EXOCHI framework depicted in Figure 1(b), the EXO architecture supports an execution model with a shared virtual address space and a POSIX multi-threaded programming model for the OS-managed IA sequencer and application-managed non-IA accelerator sequencers.

EXO differs from the existing tightly coupled approaches (category 1) by allowing independent sequencing and concurrent execution of multiple instruction streams on multiple sequencers within a single OS thread context. EXO also differs from the loosely coupled, driver-based approaches (category 2) by directly exposing the heterogeneous sequencers to application programs and by supporting a shared virtual address space amongst these sequencers. EXOCHI's user-level runtime can be used to schedule shreds and coordinate light-weight inter-shred data communication efficiently through shared virtual memory.

In addition, by supporting the shared virtual memory heterogeneous multi-threaded execution model, the CHI integrated programming environment enables the application developer to inline blocks of accelerator specific assembly or domain-specific languages within traditional C/C++ code. This allows performance sensitive parts of an algorithm to be optimized for the accelerator ISA just as Intel's SSE ISA extensions are traditionally used in implementing a high-performance math library. CHI's extensions to OpenMP allow programmers to express the underlying thread-level parallelism in a familiar parallel programming environment.

EXO ARCHITECTURE

Architecturally, EXO extends the Multiple Instruction Stream Processor (MISP) architecture [7] in three significant ways: (1) MISP exoskeleton (2) Address Translation Remapping (ATR), and (3) Collaborative Exception Handling (CEH). With this architectural support, EXO fundamentally enables a powerful shared virtual memory heterogeneous multi-threaded

programming model, despite ISA differences between the IA sequencer and the exo-sequencers.

MISP Exoskeleton

EXO provides a minimal architectural "wrapper," or exoskeleton, to make a non-IA heterogeneous accelerator sequencer conform to the MISP inter-sequencer signaling mechanism. With this exoskeleton, the accelerator sequencer can be exposed as an application-managed sequencer, even though it has a different ISA from the IA sequencers. To distinguish from an application-managed IA sequencer, we call such heterogeneous accelerator sequencers *exo-sequencers*. The exoskeleton supports interaction with the OS-managed IA sequencer through either initiating or responding to inter-sequencer user-level interrupts. With this enhancement, the code on an OS-managed IA sequencer can use MISP's **SIGNAL** instruction to dispatch shreds of a non-IA ISA to run on the exo-sequencers. This demands no additional OS support beyond MISP's requirements.

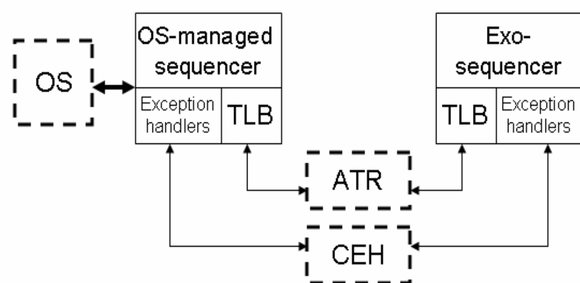

Figure 2: ATR and CEH between heterogeneous sequencers

Microarchitecture Support

Address Translation Remapping

To support shared virtual memory between the OS-managed IA sequencer and the exo-sequencers, EXO provides an ATR mechanism to allow the IA sequencer to handle page faults on behalf of the exo-sequencers.

Maintaining a shared virtual address space between two sequencers requires the same virtual address to be resolved to the same physical memory address on both sequencers. Among sequencers of the same architecture, this is accomplished by having the sequencers utilize the same page table for address translation. In a heterogeneous multi-core with IA sequencers and non-IA exo-sequencers, however, the page table format understood by each sequencer may differ. Directly accessing the IA page table is not an option for the exo-sequencers in such a case.

EXO solves this problem with its ATR mechanism. With ATR, when an exo-sequencer incurs a translation miss, it suspends shred execution and signals the IA sequencer to request *proxy execution* in order to service that Translation Lookaside Buffer (TLB) miss or page fault. Like MISP, upon receiving the proxy request as a user-level interrupt, the IA shred transfers control to a proxy handler that will touch the virtual address on behalf of the exo-sequencer. Once the page fault is serviced on the IA sequencer, however, unlike MISP, ATR will transcode the IA page table entry to the format of the exo-sequencer's page table entry before inserting the entry into the exo-sequencer's TLB. The exo-sequencer's TLB then points to the same physical page as the IA's TLB and can directly access the needed data. The exo-sequencer then resumes execution. As shown in Figure 2, an address translation remapping mechanism is responsible for remapping the IA page entry to the native format on the accelerator.

The shared virtual memory space for heterogeneous sequencers provides many benefits over the alternative approaches. It provides the essential architectural foundation to extend the classic shared memory multithreaded programming paradigm to heterogeneous multi-core processors. With a shared virtual address space, shreds from a single memory image executable running on IA sequencers and exo-sequencers can perform data communication and synchronization in familiar and efficient ways, e.g., without having to resort to explicit data copying as is necessary in the loosely-coupled approach.

It is important to note that even though ATR provides the necessary architectural support for a shared virtual address space, ATR by itself does not guarantee or require cache coherence between the IA sequencer and an exo-sequencer. In the absence of hardware support for cache coherence between the IA sequencer and an exo-sequencer, it is the responsibility of the programmer to use critical sections to protect other IA shreds from reading or writing the data being processed by shreds on the exo-sequencers. When an IA shred hands off a shared data structure to a shred on an exo-sequencer to process, the IA shred must first commit any dirty lines to main memory. Similarly, when the exo-sequencer shred completes its computation, it also needs to flush its cache before releasing a semaphore to the IA sequencer.

Clearly, with full cache coherence support between the IA sequencer and the exo-sequencer the programmer's work can be greatly eased. In particular, there is no need to use critical sections to ensure mutual exclusion on reads to the shared working set. This enables more concurrency between shreds on the IA sequencer and the exo-sequencer.

Collaborative Exception Handling

As with page faults, execution on the exo-sequencers could potentially incur exceptions or faults that require OS services. In conventional MISP, if an exception occurs on an application-managed sequencer, the instruction causing the exception can be replayed on the OS-managed sequencer through proxy execution. However, when the exception occurs on a non-IA exo-sequencer, the faulting instruction cannot simply be replayed on the IA CPU sequencer. Because the exo-sequencer uses a different ISA, the faulting instruction might have a data type that is not supported by IA ISA directly, or the exo-sequencer may require a different exception handling convention. To address this, EXO adds hardware support for CEH and a software-based exception handling mechanism, which allows faults or exceptions that occur on the exo-sequencer to be handled by the OS by proxy on the OS-managed IA sequencer.

Through CEH, an exception is handled in a similar fashion to a TLB miss. For example, as shown in Figure 2, when a double precision floating point vector instruction on an exo-sequencer incurs an exception, the exo-sequencer first signals the IA sequencer, as it does with ATR. The IA sequencer then functions as the proxy for the exo-sequencer by invoking an application-level handler to emulate the faulting vector instruction or use an OS service such as Structured Exception Handling (SEH) to provide full IEEE-compliant handling of the exception on the particular excepting scalar element. Once the exception is handled on the IA sequencer, CEH ensures the result is updated on the exo-sequencer before resuming execution.

Accelerator Exo-Sequencer: Two Examples

Media Accelerator

One example of an exo-sequencer accelerator is the integrated Intel Graphics Media Accelerator X3000 from the Intel® 965G Express chipset [9]. Figure 3 shows a high-level view of the GMA X3000 hardware. The GMA X3000 contains eight programmable, general-purpose graphics media accelerator cores, called Execution Units (EU), each of which supports four hardware thread contexts. From the programmer's perspective, 32 exo-sequencers are available. We use a custom emulation firmware that uses an IA CPU core as the OS-managed sequencer and uses the 32 GMA X3000 sequencers as exo-sequencers. The firmware implements all essential architectural extensions required by the EXO architecture, including MISP exoskeleton, ATR, and CEH.

A shred for the GMA X3000 exo-sequencer can be created either by an IA shred or spawned from another GMA X3000 shred. Once created, GMA X3000 shreds are scheduled in a software work queue in shared virtual memory like POSIX threads. The work queue can have a far greater number of shreds than the number of GMA X3000 exo-sequencers. The emulation firmware is responsible for translating a shred descriptor, which includes shred continuation information like instruction and data pointers to the shared memory, into implementation-specific hardware commands that the GMA X3000 exo-sequencers can consume and execute. The emulation layer hides all device-specific hardware details from the programmer.

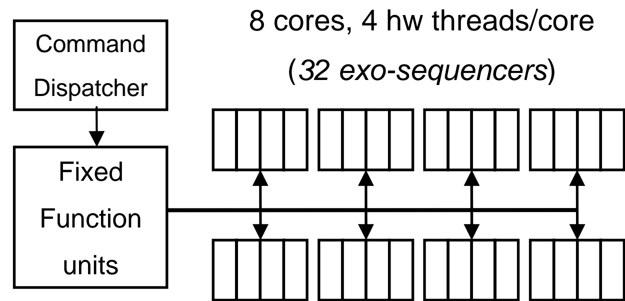

Figure 3: High-level view of the Intel GMA X3000

Communication Accelerator

Another example of the exo-sequencer accelerator is the Scalable Communication Cores (SCC) [8]. SCC is a research prototype designed for a reconfigurable radio baseband that is capable of processing several wireless standard protocols, such as WiFi, WiMax [12], or cellular infrastructure, with a common set of hardware. The SCC system architecture consists of a heterogeneous set of coarse-grained, highly optimized baseband Processing Elements (PEs).

One type of PE is the Data Processing Element (DPE) core, which performs computationally intensive operations, such as the Fast Fourier Transform (FFT) that is commonly used in many standard protocols. The DPE core structure consists of control and computation units and several memory blocks. DPE cores are connected via flexible interconnect matrices. Asynchronous data-path swap units support commutations from any of four inputs to any of four outputs. Reconfiguration of the data-path can be done dynamically with interconnection information and operation parameters stored in the configuration cache.

Inside DPE, there is a configuration (CFG) queue that is part of a special task scheduling mechanism. Each task pointer that is pushed onto the CFG queue will be fetched by the core engine. Each launched task becomes an exo-sequencer running on DPE. The DPE can be configured to use multiple CFG queues, thus implying a multi-threaded implementation. This allows multiple exo-sequencers to run concurrently on the DPE engine.

CHI PROGRAMMING ENVIRONMENT

C for Heterogeneous Integration (CHI) is designed to provide an IA look-n-feel programming environment to support user-level multi-shredding on heterogeneous sequencers. In the CHI infrastructure, we enhance the Intel C++ Compiler to support accelerator-specific inline assembly within the C/C++ source. In addition, we extend OpenMP pragmas to support heterogeneous multi-shredding and provide the related runtime support. The runtime library is responsible for judiciously scheduling heterogeneous shreds across the exo-sequencers. The compiler can also embed debugging information for different ISAs in a single binary. Such information can be used by an enhanced version of the Intel Debugger (IDB) to enable source-level debugging for both C/C++ code on the IA CPU target and the accelerator-specific code on the accelerator target. Figure 4 depicts the overall CHI compilation infrastructure. Three new capabilities are provided in the CHI compiler to allow programmers to express multi-shredded computation for the heterogeneous exo-sequencers in the C/C++ source code:

- A method to specify a region of accelerator-specific computation in either inline assembly or domain-specific language.
- A method to specify fork-join or producer-consumer style shred-level parallel execution for the inline accelerator-specific code region with OpenMP pragmas.
- A method to specify input and output memory regions and live-in values for the accelerator-specific code region.

Inline Accelerator Assembly Support

C/C++ provides a facility to inline assembly code blocks directly within the high-level source code. This capability provides programmers access to new instructions or processor features not exposed through the compiler and allows the most performance-critical parts of a program to be custom optimized in assembly. This inline assembly construct can be naturally extended to provide accelerator-specific inline assembly support.

Many variants of `asm` keyword and syntax exist. In CHI we adopt the Microsoft MASM syntax, i.e.,

`__asm` {asm_statements;}

where brackets are used to enclose the assembly statements. `__asm` is the keyword that indicates the enclosed block of code is a special assembly block written specifically for the given accelerator ISA. The *asm_statements* enclosed in the ensuing brackets are compiled into an accelerator-specific executable binary. The target ISA for the *asm_statements* is specified through the enclosing OpenMP pragma with the `target` clause, which is described in this paper in the section entitled "OpenMP Parallel Pragma Extension." As shown in Figure 4, a separate accelerator-specific assembler is dynamically linked with the Intel compiler. Figure 5 shows an example of C code using the extended OpenMP pragmas and CHI runtime APIs for a heterogeneous target consisting of an IA32 sequencer and GMA X3000 exo-sequencers.

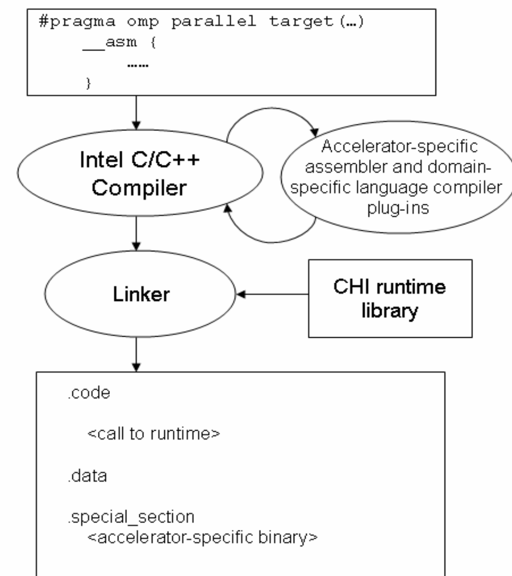

Figure 4: CHI compilation flow

Similar to traditional inline assembly, this accelerator-specific assembler generates code for the target ISA by translating the inline assembly instructions enclosed in the brackets into binary code and resolving symbolic names for memory locations and other entities referenced within the assembly block. After the assembler compiles the assembly block, the resulting binary code is embedded in a special code section of the executable indexed with a unique identifier. The final executable is a fat binary, consisting of binary code sections corresponding to different ISAs.

Domain-specific Language Support

In addition to supporting accelerator-specific inline assembly, the capability of the C/C++ compiler can be further extended to provide a facility to inline domain-specific language blocks directly within the high-level source code. These domain-specific languages are designed to utilize the accelerator-specific features not exposed through the general C/C++ programming environment. Therefore, the programmers can take advantage of the full capability of the underlying

accelerators without programming the exo-sequencer directly in assembly language.

```
int *A = malloc(n);
int *B = malloc(n);
int *C = malloc(n);

A_desc = chi_alloc_surface(A, X3000_INPUT, n, 1);
B_desc = chi_alloc_surface(B, X3000_INPUT, n, 1);
C_desc = chi_alloc_surface(C, X3000_OUTPUT, n, 1);
#pragma omp parallel target(x3000) shared(A,B,C)
    descriptor(A_desc,B_desc,C_desc) private(i)
{
  for (i=0; i<n/8; i++)
     __asm
     {
        shl.1.w    vr1 = i, 3
        ld.8.dw    [vr2..vr9] = (A, vr1, 0)
        ld.8.dw    [vr10..vr17] = (B, vr1, 0)
        add.8.dw   [vr18..r25] = [vr2..vr9], [vr10..vr17]
        st.8.dw    (C, vr1, 0) = [vr18..vr25]
     }
}
#pragma omp parallel for shared(D,E,F) private(i)
{
   for (i=0; i<n; i++)
      F[i] = D[i] + E[i];
}
```

Figure 5: Example GMA X3000 inline assembly

To provide a uniform programming interface to programmers, we adopt the format similar to that of the `asm` syntax, i.e.,

`__<language keyword>` {domain-specific language statements;}

where brackets are used to enclose the domain-specific language statements. `__<language keyword>` can be any language that is supported by CHI. Upon parsing the particular language keyword, the C/C++ compiler invokes the corresponding domain-specific compiler plug-ins to generate the accelerator-specific binary, similar to how it is done with the inline assembly support as described in the section entitled "Inline Accelerator Assembly Support."

Figure 6 shows an example of the domain-specific language support to the Data-stream Programming Language (DPL) that is specifically designed for the retargetable SCC-DPE accelerator. DPL provides essential high-level functions to exploit the inner microarchitecture of the DPE systolic arrays. The programmers can embed DPL code within the brackets preceded by the `__dpl` keyword.

OpenMP Parallel Pragma Extension

CHI extends the OpenMP `parallel` pragma. The construct for generating heterogeneous shreds of an accelerator-specific instruction set is outlined in Figure 7(a). The `target` clause specifies the particular accelerator instruction set used within the parallel region. The compiler inserts appropriate calls to the CHI runtime layer to enable judicious dynamic shred scheduling and dispatching onto the targeted exo-sequencers. When the main IA shred encounters an accelerator-specific `parallel` construct with the `target(`*`targetISA`*`)` clause, the IA shred spawns a team of `num_threads` heterogeneous shreds for the parallel region, where each shred eventually executes the enclosed assembly block on an exo-sequencer.

```
float Vin[4];
float Vout[4];

void *in_desc  = (void *)chi_alloc_buffer_desc
          (DPE_INPUT_BUFFER,  Vin, 4, 1);
void *out_desc = (void *)chi_alloc_buffer_desc
          (DPE_OUTPUT_BUFFER, Vout, 4, 1);

#pragma omp parallel target(dpe)
       shared(Vin,Vout)
descriptor(in_desc,out_desc)
{
  __dpl {
    configuration[1] cfgMult( vector val[1],
                    vector coeff[1] )
    {
      result bs( mull(val, coeff), 13 );
    }
    flow[4] multiFlow( vector vec[4],
           vector coeffs[4])
    {
      vector ret[4]; result out;
      selector[iter : 4] sel[1] = {{ iter }};
      selector[iter : 4] selRev[1] = {{ 3 - iter
}};
      ret[sel] = cfgMult(vec[sel], coeffs[selRev]);
    }
    vector cf[4] = { 0.5 + I * 0.0 };
    program dlMain()
    {
      Vout = multiFlow(Vin, cf);
    }
  }
}
```

Figure 6: Example inline DPL code using CH

By default, the main IA shred waits at the end of the construct until it is notified by the CHI runtime of the completion of all heterogeneous shreds. Similar to the traditional `nowait` clause, an optional `master_nowait` clause allows the main IA shred to continue execution past the construct after spawning the team of heterogeneous shreds, without having to wait for their completion. This allows concurrent execution on both the IA sequencer and its exo-sequencers. The CHI runtime is responsible for asynchronously notifying the IA sequencer of the eventual completion of all heterogeneous shreds.

OpenMP Work-Queuing Extension

In order to support concurrent threads with intricate dynamic inter-thread dependencies (e.g., due to the use of irregular data structures), the Intel C++ Compiler supports irregular parallelism through two special OpenMP pragmas, `taskq` and `task` [23]. In CHI, we further enhance the compiler and runtime to support inter-shred dependencies among heterogeneous shreds using these pragmas. The `parallel taskq` construct and the `task` construct for an exo-sequencer are outlined in Figure 7(b) and Figure 7(c).

```
#pragma omp parallel target(targetISA) [clause[[,]clause]…]
                structured-block
```

Where clause can be any of the following:

```
firstprivate(variable-list)
private(variable-list)
shared(variable-ptr-list)
descriptor(descriptor-ptr-list)
num_threads(integer-expression)
master_nowait
```

(a) Parallel specification in fork-join threading model

```
#pragma intel omp taskq target(targetISA) [clause[[,]clause]…]
                structured-block
```

Where clause can be any of the following:

```
firstprivate(variable-list)
private(variable-list)
shared(variable-ptr-list)
descriptor(descriptor-ptr-list)
num_threads(integer-expression)
master_nowait
```

(b) Queue specification in producer-consumer threading model

```
#pragma intel omp task target(targetISA) [clause[[,]clause]…]
                structured-block
```

Where clause can be any of the following:

```
captureprivate(variable-list)
shared(variable-ptr-list)
descriptor(descriptor-ptr-list)
```

(c) Task specification in producer-consumer threading model

Figure 7: CHI extensions to OpenMP pragmas

CHI Runtime Support

The CHI runtime is a software library that translates the programmer-specified OpenMP directives into primitives to create and manage shreds that can carry out parallel execution on the heterogeneous multi-core target. Like conventional OpenMP runtimes, the CHI runtime layer provides a layer of abstraction that hides the details of managing the exo-sequencers from the programmer.

In order to allow the accelerator more efficient access to the C/C++ variables specified by the `shared` data clause, programmers can use the CHI runtime APIs to convey accelerator-specific access information through data structures known as descriptors. Descriptors are used by the accelerator to interpret the attributes of the `shared` variables that are accessed by the shreds.

EXOCHI PROTOTYPE

The EXOCHI framework described in this paper has already been deployed within Intel for successful development of production-quality, GMA X3000 media-processing kernels and other workloads of growing importance [2]. Figures 8 and 9 provide examples of the use of how an IA look-n-feel allows familiar development tools and environments to be used in writing heterogeneous multi-shredded code. Figure 8 shows the use of familiar legacy development tools (Microsoft Visual Studio*) for development and debugging of heterogeneous multi-shredded code. Figure 9 illustrates the compilation and execution of such a program.

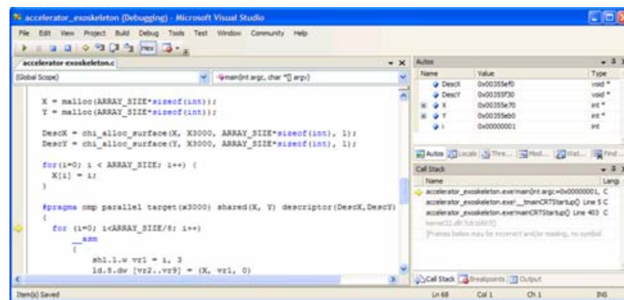

Figure 8: IA Look-n-Feel IDE (Microsoft Visual Studio) for application development

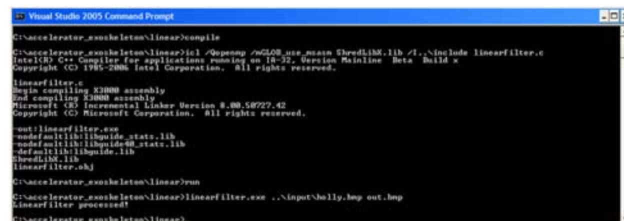

Figure 9: IA Look-n-Feel compilation and execution

Performance Evaluation

To evaluate the performance of our EXOCHI prototype we select a representative subset of the kernels that have been developed. These kernels exhibit a significant amount of data- and thread-level parallelism and thus, readily lend themselves to efficient execution on the GMA X3000 exo-sequencers.

Implementation of these kernels is made easy due to special GMA X3000 ISA features optimized for media processing. The key ISA features include wide SIMD instructions, predication support, and a large register file of 64 to 128 vector registers for each GMA X3000 exo-sequencer. With CHI, programmers can directly use the GMA X3000 ISA features via inline assembly in C/C++ code as if they are traditional ISA extensions to IA, such as SSE. By providing such IA look-n-feel, CHI enables highly productive development of heterogeneous multi-shredded code.

All benchmarks are compiled with the enhanced version of the Intel C++ Compiler using the most aggressive optimization settings (–fast –Qprof_use). These compiler optimizations include auto-vectorization, profile-guided optimization, and tune specifically for the Intel Core 2 Duo processor used in the EXO prototype system. `LinearFilter`, `SepiaTone` and `FGT` make use of the optimized and SSE-enhanced Intel IPP library, and the other benchmarks were manually tuned and SSE-optimized. Performance results measure the wall clock execution time.

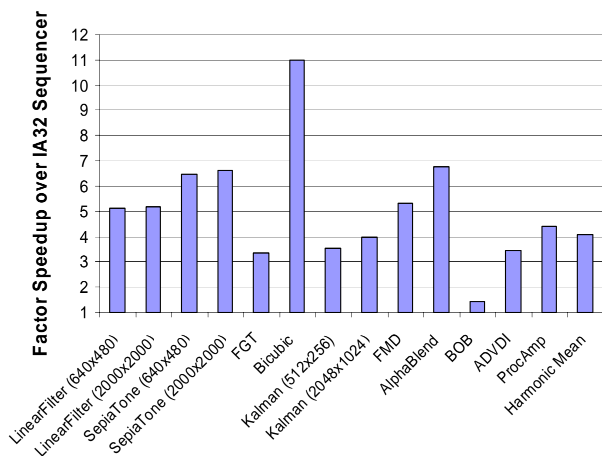

Figure 10: Speedup from execution on GMA X3000 exo-sequencers over IA sequence

Performance Speedup on GMA X3000 Exo-sequencers over IA Sequencer

Figure 10 shows the speedup achieved over IA sequencer execution by executing media kernels on the GMA X3000 exo-sequencers. Significant speedup is achieved, ranging from 1.41X for BOB up to 10.97X for Bicubic. Two factors are crucial in achieving this high throughput performance on the GMA X3000 exo-sequencers. Most important is the availability of abundant shred-level parallelism. As each GMA X3000 exo-sequencer supports only in-order execution within a shred, the accelerator relies on the presence of multiple concurrent shreds to cover up stalls incurred in one shred by switching to another shred. A second, but related issue, is the need to maximize cache hit rate and memory bandwidth utilization. The GMA X3000 supports simultaneous execution of 32 hardware threads, each of which might be reading and writing multiple data streams. The CHI runtime allows programmers to carefully orchestrate shred scheduling to ensure shreds accessing adjacent or overlapping macroblocks are ordered closely together in the work queue so as to take advantage of spatial and temporal localities.

Other than support for thread-level parallelism, the GMA X3000 ISA also provides strong support for data-level parallelism. It features significantly wider SIMD operations (8- to 16-wide vector) than the SSE on today's IA CPU.

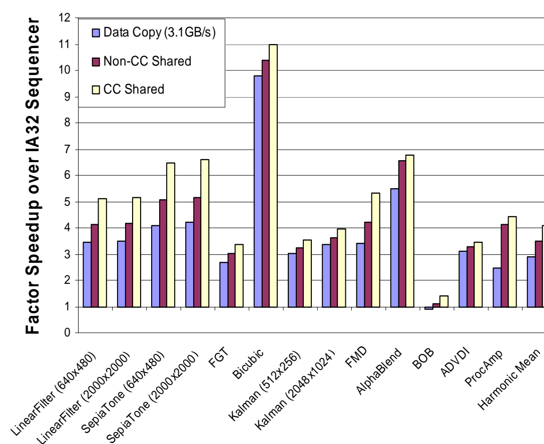

Figure 11: Impact of shared virtual memory

Impact of Data Copying Versus Shared Virtual Address Space

In general, the performance improvement achieved by using an accelerator is determined not only by the accelerator architecture but also by the overhead of data communication between the CPU and accelerator. This overhead varies greatly depending on the memory model between the CPU and the accelerator. Figure 11 shows overall performance improvement achieved with a cache coherent shared virtual memory model between the IA sequencer and the GMA X3000 exo-sequencers. In the absence of cache coherence or shared memory, the data communication overhead can significantly degrade the speedup achieved from accelerating the computation. In Figure 11 we contrast performance impacts for three memory model configurations.

The first configuration, *Data Copy*, assumes a model without shared virtual memory and no cache coherence between the IA sequencer and the GMA X3000 exo-sequencers. Consequently, data communication between IA shred and GMA X3000 shreds requires explicit data copying, for which we assume a 3.1GB/s data copy rate. This corresponds to an aggressive data copy rate using an SSE-enhanced memory copy routine when copying data from a cacheable memory source to a destination region marked as uncacheable, write-combining memory. The Intel Core 2 Duo processor features special write-combining buffers that allow aggressive burst mode transfers when copying from cacheable memory to write-combining memory. Due to the lack of shared virtual memory, the inter-shred communication between the IA shred and GMA X3000 shreds resembles that of traditional message passing communication between processes from different address spaces.

The second configuration, *Non-CC Shared*, assumes a shared virtual address space but without cache coherency between the IA sequencer an the GMA X3000 exo-sequencers. Data copying can be avoided in this case as both the IA sequencer and GMA X3000 exo-sequencers can access the identical physical memory location for the same virtual address. Memory writes performed by the IA sequencer or the GMA X3000 exo-sequencers may not be visible to the other until after a cache flush operation, which forces any dirty cache lines to be written back to main memory. However, data communication can still be accomplished by passing a pointer to a shared data structure between the IA sequencer and a GMA X3000 exo-sequencer as long as cache flush operations are appropriately invoked. Due to the lack of cache coherence, the IA shred and the GMA X3000 shreds need to use critical sections to enforce mutually exclusive access to shared data structures. The semaphore on the critical section will not be released until the GMA X3000 exo-sequencers completely flush the dirty lines to memory.

The first configuration, *Data Copy*, assumes a model without shared virtual memory and no cache coherence between the IA sequencer and the GMA X3000 exo-sequencers. Consequently, data communication between IA shred and GMA X3000 shreds requires explicit data copying, for which we assume a 3.1GB/s data copy rate. This corresponds to an aggressive data copy rate using an SSE-enhanced memory copy routine when copying data from a cacheable memory source to a destination region marked as uncacheable, write-combining memory. The Intel Core 2 Duo processor features special write-combining buffers that allow aggressive burst mode transfers when copying from cacheable memory to write-combining memory. Due to the lack of shared virtual memory, the inter-shred communication between the IA shred and GMA X3000 shreds resembles that of traditional message passing communication between processes from different address spaces.

The third configuration, *CC Shared*, models a cache-coherent shared virtual address space, which is the configuration assumed in Figure 10. In this model, data communication between the IA shred and the GMA X3000 shreds becomes much more efficient. Similarly, the synchronization on mutual access to shared data structure is also made much easier for programmers. For example, while critical sections are still necessary to provide mutual exclusion on writes to a shared variable, one shred can always read the shared variables that are updated by the other shreds. This allows more execution concurrency between shreds.

The performance data in Figure 11 demonstrate the benefits of a shared virtual address space compared to data copying. While significant performance improvement is still possible even with data copying, for computationally intensive kernels (e.g., `bicubic` and `ADVDI`), the gains are significantly reduced from the original *CC Shared* configuration in cases such as `LinearFilter` and `BOB`. For benchmarks in which the GMA X3000 performs little computation on the loaded input data, the time to copy data between separate address spaces represents a significant fraction of the processing time. Even with a highly optimized implementation on the latest IA Intel Core 2 Duo processor, the data copying achieves only 70.5% of that seen for a coherent shared virtual address space.

The cost of copying data can be ameliorated if the IA sequencer and the GMA X3000 exo-sequencers operate within a shared virtual address space, even if cache coherency is not supported. The time required to flush caches is still nontrivial, however, and the lack of coherency (*Non-CC Shared*) still yields 85.3% of the performance achieved with full cache coherency. Support for cache coherence improves performance because the cache flush operation is not needed to synchronize memory accesses.

For the *Non-CC Shared* configuration, when an IA shred spawns GMA X3000 shreds, it may appear necessary to flush the IA sequencer's cache fully before any GMA X3000 shred can be launched. In reality the majority of the cache flush operation on the IA sequencer can be overlapped with parallel shred execution on the GMA X3000 exo-sequencers if cache flush operations and shred launches can be interleaved. As each exo-sequencer shred only reads and writes a tiny portion of each data buffer (e.g., a 16 pixel by 16 pixel macroblock), as long as those data have been flushed back to memory by the IA producer shred, the exo-sequencer consumer shred for that macroblock can be launched and can execute safely. Additional cache flush operations can then proceed in parallel with useful work being performed in parallel on the exo-sequencers.

CONCLUSION

In this paper we present the EXO MIMD extension to the IA ISA to expose heterogeneous cores as application-level architecture resources and provide shared virtual memory to support the classic multi-shredded programming model for heterogeneous multi-core processors. The EXO architecture allows application programs to directly use heterogeneous hardware as MIMD functional units while requiring minimal additional dependency on the existing OS ecosystem. In addition, in order to take advantage of the rich ecosystem legacy for IA software development, the CHI programming environment provides an IA look-n-feel by

extending the Intel C++ Compiler, OpenMP runtime, and debugger toolchains to support user-level heterogeneous multi-shredding. Since its development, EXOCHI has been used in Intel's production media kernel development. Based on extensive feedback from developers, there is strong evidence that the IA look-n-feel of the programming environment has significantly improved productivity over prior device driver-based development environments.

ACKNOWLEDGMENTS

We thank Nick Yang, Porus Khajotia, Prasoonkumar Surti, Bob Dreyer, Sang-hee Lee, Katen Shah, Mike Dwyer, Yi-jen Chiu, Lian Tang, Igor Kozintsev, Xintian Wu, Bevin Brett, Susan Macchia, Ping Liu, Nenad Ukropina, Todd Schwartz, Jenny Nieh, David Sehr, Wei Li, and Sanjiv Shah for the productive collaboration throughout the EXOCHI project. We also appreciate the support from Shekhar Borkar, Joe Schutz, Tom Piazza, Justin Rattner, Jim Held, Steve Pawlowski, Kevin J. Smith, Bill Savage, Ketan Paranjape, Raj Hazra, Alan Crouch, Bryant Bigbee, Wilf Pinfold, Dave Shinsel, Ajay Bhatt, Doug Carmean, Per Hammarlund, Dion Rodgers, Steve Whalley, Avi Mendelson, and Prashant Sethi. In addition, we thank Anne Bracy, Ethan Schuchman, Ankur Khandelwal, Marian Lacey, and the anonymous reviewers whose valuable feedback has helped the authors greatly improve the quality of this paper.

REFERENCES

[1] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *ACM Transactions on Graphics*, 23(3): 777–786, 2004.

[2] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology@Intel Magazine*, February 2005.

[3] *GLSL–OpenGL Shading Language*, in www.wikipedia.org/wiki/GLSL*

[4] R. Gonzalez, "A Software-configurable Processor Architecture," *IEEE Micro,* Sept./Oct. 2006, pp. 42–51.

[5] *GPGPU: General Purpose Computation using Graphics Hardware*, at www.gpgpu.org*

[6] E. Grochowski, M. Annavaram, "Energy per Instruction Trends," in *Intel® Microprocessors. Technology@Intel Magazine*, March 2006, at http://www.intel.com/technology/magazine/research/energy-per-instruction-0306.pdf

[7] R. Hankins, G. Chinya, J. Collins, P. Wang, R. Rakvic, H. Wang and J. Shen, "Multiple Instruction Stream Processor, in *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[8] J. Hoffman, D. A. Ilitzky, A. Chun, A. Chapyzhenka, "Architecture of Scalable Communication Core," in *First International Symposium on Networks-on-Chip*, 2007.

[9] Intel Corp., *Intel G965 Express Chipset*, at http://www.intel.com/products/chipsets/g965/prod_brief.pdf

[10] Intel Corp., "Intel's Next Generation Integrated Graphics Architecture – Intel Graphics Media Accelerator X3000 and 3000," *White Paper*, 2006.

[11] Intel Corp., "Tera-scale Research Prototype: Connecting 80 Simple Sores on a Single Test Chip," ftp://download.intel.com/research/platform/tera-scale/tera-scaleresearchprototypebackgrounder.pdf

[12] Intel Corp., "WiMAX," in *Intel Technology Journal* Vol. 8 Issue 3, at ftp://download.intel.com/technology/itj/2004/volume08issue03/vol8_iss03.pdf.

[13] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. Ahn, P. Mattson and J. Owens, "Programmable Stream Processors," in *IEEE Computer*, 2003.

[14] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single-ISA Heterogeneous Multi-Core Architectures for Multi-threaded Workload Performance," in *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[15] F. Labonte, P Mattson, W. Thies, I. Buck, C. Kozyrakis, and M. Horowitz, "The Stream Virtual Machine," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[16] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language*," ACM Transactions on Graphics* 22, 3, 896–907.

[17] M. McCool and S. Toit, *Metaprogramming GPUs with Sh*, A K Peters, Ltd., Wellesley, MA, 2004.

[18] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Eurographics*, August 2005.

[19] PeakStream Inc., "The PeakStream Platform: High Productivity Software Development for Multi-core Processors," *White Paper*, 2006.

[20] RapidMind Inc., "Performance Evaluation of GPUs using the RapidMind Development Platform," *Supercomputing'06*.

[21] S. Shah, G. Haab, P. Petersen, J. Throop, "Flexible control structures for parallelism in OpenMP," in

Proceedings of the First European Workshop on OpenMP, Sept. 1999.

[22] M. Segal and M. Peercy, "A Performance-Oriented Data Parallel Virtual Machine for GPUs," *SIGGRAPH,* 2006.

[23] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen, "Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures," in *EWOMP*, 2002.

[24] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," in *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[25] W. Thies, M. Karczmarek, and S. Amarasinghe,"StreamIt: A Language for Streaming Applications," *CC,* 2002.

[26] X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen, "Compiler and Runtime Support for Running OpenMP Programs on Pentium- and Itanium-Architectures," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, April 2003.

[27] O. Wechsler, "Inside Intel Core Microarchitecture: Setting New Standards for Energy-efficient Performance," *Technology@Intel Magazine*, 2006.

[28] D. Zhang, Z Li, H. Song, and L. Liu, "A Programming Model for an Embedded Media Processing Architecture," *SAMOS*, 2005.

AUTHORS' BIOGRAPHIES

Perry Wang is a Senior Staff Engineer with Intel's Corporate Technology Group. His work involves research on processor architecture, microarchitecture and compiler optimization techniques. Perry has been with Intel for 12 years and holds a master's degree in Computer Engineering from the University of Michigan.

Jamison Collins is a Staff Engineer with Intel's Corporate Technology Group. His work involves exploring and prototyping future Intel processor architecture and microarchitecture. Jamison has been with Intel for four years and holds a Ph.D. degree in Computer Science and Engineering from UC San Diego.

Gautham Chinya is a Senior Staff Engineer with Intel's Corporate Technology Group. His work involves exploring future processor system architecture and interaction with operating systems. Gautham has been with Intel for eight years and holds a master's degree in Computer Engineering from Purdue University.

Hong Jiang is a Senior Principal Engineer with Intel's Mobility Group. He is Intel's lead architect specializing in video technology. Hong has been with Intel for ten years and holds a Ph.D. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign.

Xinmin Tian is a Principal Engineer with Intel's Software Solutions Group. He is Intel's lead compiler architect specializing in compiler parallelization, OpenMP, vectorization, and transactional memory compiler development projects. Xinmin has been with Intel for eight years and holds a Ph.D. degree in Computer Science from Tsinghua University.

Milind Girkar is a Principal Engineer with Intel's Software Solutions Group. He is Intel's lead compiler architect specializing in compiler parallelization and is responsible for planning the compiler requirements for future Intel processors. Milind has been with Intel for twelve years and holds a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign.

Lisa Pearce is the software engineering manager with Intel's Mobility Group responsible for media development and content protection for all Intel integrated graphics solutions. Lisa has been with Intel for ten years and holds a bachelor's degree in Computer Science from Virginia Tech.

Guei-yuan Lueh is a Principal Engineer with Intel's Mobility Group. He leads the development of advanced compiler and runtime technology for Intel graphics solutions. Guei-yuan has been with Intel for 10 years and holds a Ph.D. degree in Computer Science from Carnegie Mellon University.

Sergey Yakoushkin is a Software Engineer in the Intel Corporate Technology Group. His work involves the development of software tools for emerging embedded platforms for communication acceleration, hardware-software co-design, and language design for data-streaming processing systems. Sergey has been with Intel for two years and holds an honours MS degree in Computer Science from St. Petersburg State University.

Hong Wang is a Senior Principal Engineer with Intel's Corporate Technology Group. His work involves research on future processor architecture and microarchitecture. Hong has been with Intel for twelve years and holds a Ph.D. degree in Electrical Engineering from the University of Rhode Island.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo,

Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

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

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS® Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from http://www.intel.com.

Additional legal notices at: http://www.intel.com/sites/corporate/tradmarx.htm.

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2007 Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information visit: www.intel.com/sites/corporate/tradmarx.htm