EXOCHI: Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System

Perry H. Wang[1], Jamison D. Collins[1], Gautham N. Chinya[1], Hong Jiang[2], Xinmin Tian[3]
Milind Girkar[3], Nick Y. Yang[2], Guei-Yuan Lueh[2], and Hong Wang[1]

Microarchitecture Research Lab, Microprocessor Technology Labs, Intel Corporation[1]
Graphics Architecture, Chipset Group, Intel Corporation[2]
Intel Compiler Lab, Software Solutions Group, Intel Corporation[3]

Contact: perry.wang@intel.com

Abstract

Future mainstream microprocessors will likely integrate specialized accelerators, such as GPUs, onto a single die to achieve better performance and power efficiency. However, it remains a keen challenge to program such a heterogeneous multi-core platform, since these specialized accelerators feature ISAs and functionality that are significantly different from the general purpose CPU cores. In this paper, we present EXOCHI: (1) *Exoskeleton Sequencer* (EXO), an architecture to represent heterogeneous accelerators as ISA-based MIMD architecture resources, and a shared virtual memory heterogeneous multithreaded program execution model that tightly couples specialized accelerator cores with general purpose CPU cores, and (2) *C for Heterogeneous Integration* (CHI), an integrated C/C++ programming environment that supports accelerator-specific inline assembly and domain-specific languages. The CHI compiler extends the OpenMP pragma for heterogeneous multithreading programming, and 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.

We have prototyped the EXO architecture on a physical heterogeneous platform consisting of an Intel® Core™ 2 Duo processor and an 8-core 32-thread Intel® Graphics Media Accelerator X3000. In addition, we have implemented the CHI integrated programming environment with the Intel® C++ Compiler, runtime toolset, and debugger. On the EXO prototype system, we have enhanced a suite of production-quality media kernels for video and image processing to utilize the accelerator through the CHI programming interface, achieving significant speedup (1.41X to 10.97X) over execution on the IA32 CPU alone.

Categories and Subject Descriptors C.1.4 [*Processor Architectures*]: Parallel Architectures; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Performance, Design, Languages

Keywords Heterogeneous multi-cores, GPU, OpenMP

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006. . . $5.00

1. 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 [14]. Fundamentally, to scale multi-core processor designs to incorporate a large number of cores, *ultra low EPI* (Energy Per Instruction) cores are essential [10]. For example, to achieve a 20X improvement (*e.g.*, from 5GOPS to 100GOPS) in throughput performance while staying below the power envelope of 150W, the building-block cores must have an average EPI of approximately 1nJ. The EPI for the Intel® Core™ 2 Duo processor core [31] is approximately 10nJ while the EPI for the 8-core 32-thread Intel® Graphics Media Accelerator X3000 [15] is only 0.3nJ. 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 [2, 17]. The key challenge then becomes how to accomplish such heterogeneous integration and achieve high performance while still maintaining the look-and-feel of the classic mainstream IA32-based programming models and software ecosystem.

In this paper, we present EXOCHI: *Exoskeleton Sequencer* (EXO), an architecture 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 multithreaded programming model for heterogeneous cores. Like application-managed *sequencers* in the Multiple Instruction Stream Processor (MISP) architecture [11], the non-IA32 cores are architecturally exposed to the programmer as a new form of sequencer resource. They can be regarded essentially 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 IA32 sequencer and accelerator sequencers facilitates code and data sharing and harmonizes cooperation between the concurrent shreds of different ISAs.

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 [25, 26, 30] to allow the programmer to express thread-level parallelism by demarcating parallel regions of code targeting non-IA32 accelerators. The CHI extensions to OpenMP support both fork-join and producer-consumer parallelism among the accelerator shreds and between the IA32

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.

This paper makes the following contributions:

- We describe the EXO architecture and the CHI programming environment that support shared virtual memory multithreaded programs for a heterogeneous multi-core processor.
- We detail a heterogeneous multi-core prototype of the EXO architecture consisting of an Intel® Core™ 2 Duo [31] processor and an 8-core 32-thread Intel® Graphics Media Accelerator (GMA) X3000 [15].
- We present an implementation of the CHI programming environment based on the Intel® C++ Compiler [29] that supports the seamless integration of accelerator-specific assembler and domain-specific languages.
- We report significant performance gains for a set of production-quality media-processing kernels by employing heterogeneous shreds on the GMA X3000, achieving speedups of up to 10.97x.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the EXO architecture and describes a physical prototype system using an Intel Core 2 Duo processor and an Intel GMA X3000. Section 4 presents the CHI integrated programming environment and details the extensions to the OpenMP pragmas for heterogeneous multithreading support in the Intel C++ Compiler and the supporting CHI runtime software system. Section 5 evaluates the performance of CHI-enabled media-processing kernels on the EXO hardware prototype. Section 6 concludes.

2. Related Work

There has been a rich body of research on heterogeneous acceleration. In most published work, the execution models usually fall into two categories: (1) an ISA-based tightly-coupled approach, or (2) a device driver-based loosely-coupled execution model. An example of the tightly-coupled approach is the SCP architecture [7] 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 [9, 22]. GPGPU uses the massive computational power of a modern GPU, normally dedicated to render graphics operations, to accelerate general purpose computation. In this line of work, depicted in Figure 1(a), the CPU resources (cores and memory) are managed by the OS, and the GPU resources are separately managed by vender-supplied device drivers. Applications and device drivers run in separate address spaces, and consequently, the data communication and synchronization between them are 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 multithreaded programming model for the OS-managed IA32 sequencer and application-managed non-IA32 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. Without requiring an OS device driver to manage the accelerators, EXOCHI's user-level runtime can be used to schedule shreds and coordinate light-weight inter-shred data communication efficiently through shared virtual memory. With direct architectural support for shared virtual memory, EXOCHI avoids the significant burden of emulating shared memory in software and orchestrating DMA data transfers to provide the illusion of a shared memory multithreaded programming model, as with CELL [5].

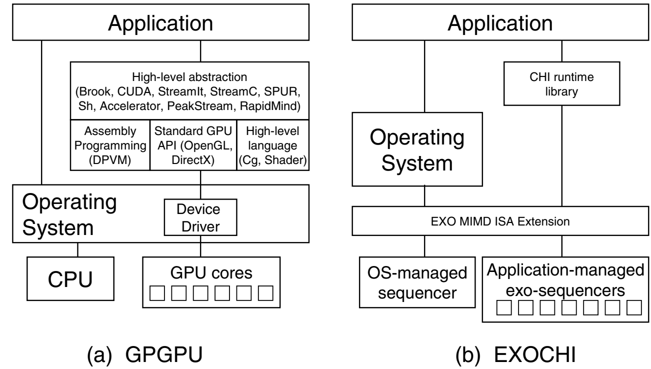

Figure 1. Alternate Programming Environments

Clearly, the programming models for most prior research on heterogeneous acceleration have been significantly influenced by the underlying architecture. For example, different levels of GPU abstractions have been introduced in various GPGPU programming models. These abstractions range from allowing to-the-metal programming in the GPU's assembly language, to high-level language extensions with full-fledged runtime systems that abstract away any implementation or vendor-specific GPU hardware intricacy. As illustrated in Figure 1(a), at the lowest level of abstraction, an infrastructure like the Data Parallel Virtual Machine (DPVM) [24] allows developers to program the GPU in its native assembly language by exposing the instruction set and by providing a set of special device driver APIs. However, DPVM exposes the GPU as a device that operates in a separate address space and requires the programmer to explicitly manage the GPU's device-specific hardware. Therefore, data communication between the application program and the GPU driver code still occurs through data copying.

In other approaches, the GPU can also be programmed using domain-specific APIs, such as OpenGL or DirectX, or through a higher-level domain-specific programming language, such as the OpenGL Shader Language [6] or Cg [19]. This approach provides a higher level abstraction of the GPU by hiding both the internal ISA and the device-specific GPU hardware complexity. However, the APIs and shader language restrictions predefined for the graphics processing domain may become cumbersome when used to express an algorithm in another domain. For example, in order to attain GPU acceleration, the programmer needs to remap the data structure representation of the application to the native data types for OpenGL and DirectX, *e.g.*, vertices and pixels. The programmer also needs to convert the algorithm into a computationally-equivalent GPU operation, *e.g.*, mapping an integer sorting operation to a texture sampling operation.

At the next level of abstraction are the streaming programming models, where the GPU is abstracted as a stream processor; such an approach is used in Brook [1], CUDA [3], StreamIt [28], StreamC [16], SPUR [32], and Sh [20]. With this approach, the programmer is presented with more general purpose language syntax

to support data structures like those in high level languages such as C/C++. While this approach provides the programmer a much more productive programming environment to produce code for the GPU hardware, the GPU still works as a device. As examples of the loosely-coupled execution model (category 2), these streaming programming environments also lack efficient integration with the traditional CPU-based programming environment and ecosystem.

Further up the abstraction hierarchy are domain-specific virtual machines, such as Stanford Stream Virtual Machine [18] and RapidMind Streaming Execution Manager [21] for stream programming, Microsoft Accelerator [27] for data parallelism exploration, and PeakStream Virtual Machine [23] for HPC acceleration. By using GPU-optimized math library APIs provided by this approach, the programmer does not need to program the GPU directly, but can still take advantage of GPU acceleration.

All these prior works treat the CPU and GPU as essentially separate, loosely-coupled entities until the highest level of abstraction is reached, as illustrated in Figure 1(a). EXO, in contrast, tightly couples the heterogeneous sequencers with the OS-managed IA32 sequencer and can potentially provide a much leaner software runtime stack for better performance, as illustrated in Figure 1(b). In addition, by supporting the shared virtual memory heterogeneous multithreaded execution model, the CHI integrated programming environment facilitates 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.

3. EXO Architecture

Architecturally, Exoskeleton Sequencer (EXO) extends the MISP architecture [11] 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 multithreaded programming model, despite ISA differences between the IA32 sequencer and the exo-sequencers.

3.1 MISP Exoskeleton

EXO provides a minimal architectural "wrapper", or exoskeleton, to make a non-IA32 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 IA32. To distinguish from an application-managed IA32 sequencer, we call such heterogeneous accelerator sequencers *exo-sequencers*. The exoskeleton supports interaction with the OS-managed IA32 sequencer through either initiating or responding to inter-sequencer user-level interrupts. With this enhancement, the code on an OS-managed IA32 sequencer can use MISP's `SIGNAL` instruction to dispatch *shreds* of a non-IA32 ISA to run on the exo-sequencers. This demands no additional OS support beyond MISP's requirements.

3.2 Address Translation Remapping

To support shared virtual memory between the OS-managed IA32 sequencer and the exo-sequencers, EXO provides an address translation remapping (ATR) mechanism to allow the IA32 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 IA32 sequencers and non-IA32 exo-sequencers however, the page table format understood by each sequencer may differ. Directly accessing the IA32 page table is not an option for the exo-sequencers in such cases. For example, the internal TLB of the Intel GMA X3000 assumes the industry standard GPU driver-oriented page table format, which is different from the IA32 page table formats. Without significant hardware changes, the exo-sequencers cannot use the IA32 page table to service TLB misses.

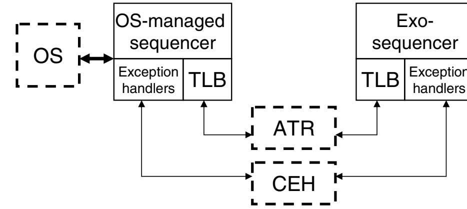

Figure 2. ATR and CEH between Heterogeneous Sequencers

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 IA32 sequencer to request *proxy execution* [11] in order to service that TLB miss or page fault. Like MISP, upon receiving the proxy request as a user-level interrupt, the IA32 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 IA32 sequencer, however, unlike MISP, ATR will transcode the IA32 page table entry to the format of the exo-sequencer's page table entry before inserting the entry to the exo-sequencer's TLB. The exo-sequencer's TLB will point to the same physical page as the IA32'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 IA32 page entry to the native format on the accelerator.

The shared virtual memory space for heterogeneous sequencers provides many benefits over the alternative approaches described in Section 2. 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 IA32 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 IA32 sequencer and an exo-sequencer. In the absence of hardware support for cache coherence between the IA32 sequencer and an exo-sequencer, it is the responsibility of the programmer to use critical sections to protect other IA32 shreds from reading or writing the data being processed by shreds on the exo-sequencers. When an IA32 shred hands off a shared data structure to a shred on an exo-sequencer to process, the IA32 shred must first flush its cache to 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 IA32 sequencer.

Clearly, with full cache coherence support between the IA32 sequencer and the exo-sequencer, the programmer's work can be

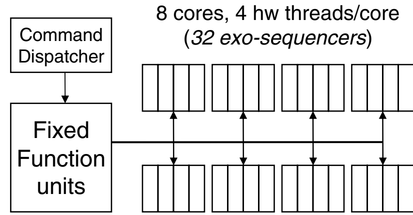

Figure 3. Intel Graphics Media Accelerator X3000

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 IA32 sequencer and the exo-sequencer. In Section 5.2, we provide a more quantitative analysis of the benefit of shared virtual memory support and the impact of cache coherence.

3.3 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-IA32 exo-sequencer, the faulting instruction cannot simply be replayed on the IA32 CPU sequencer. Because the exo-sequencer uses a different ISA, the faulting instruction might have a data type that is not supported by IA32 ISA directly, or the exo-sequencer may require a different exception handling convention. To address this, EXO adds hardware support for collaborative exception handling (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 IA32 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 IA32 sequencer, as it does with ATR. The IA32 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 IA32 sequencer, CEH ensures the result is updated in the exo-sequencer before resuming execution.

3.4 EXO Hardware Prototype

We prototyped the EXO architecture enhancements using an Intel Santa Rosa platform [13]. The system consists of an Intel Core 2 Duo [31] processor and an Intel 965G Express chipset [12], which contains the integrated Intel Graphics Media Accelerator X3000 [15]. 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 IA32 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 IA32 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.

On the GMA X3000, one shred can write directly to another shred's register file to facilitate inter-shred communication. This creates a producer-consumer relationship between shreds and enables the development of very sophisticated yet efficient multithreaded algorithms. The X3000 ISA is optimized for data- and thread-level parallelism and each exo-sequencer supports wide SIMD operations on up to 16 data elements in parallel. The X3000 ISA also features both specialized instructions for media processing and a full complement of control flow mechanisms. The exo-sequencers share access to specialized, fixed function hardware that can execute performance-critical tasks, such as texture sampling and scattering/gathering memory operations.

The four exo-sequencers, physically implemented in each GMA X3000 core, alternate fetching through fly-weight switch-on-stall multithreading. As each exo-sequencer fetches and retires instructions in-order, the core's fine-grained thread multiplexing capability plays a critical role in sustaining throughput performance. For example, a stall in one exo-sequencer due to an instruction dependency or cache miss is mitigated by preferentially fetching from one of the other three exo-sequencers bound to the same core.

4. CHI Programming Environment

C for Heterogeneous Integration (CHI) is designed to provide an IA32 look-and-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 IA32 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.

4.1 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.

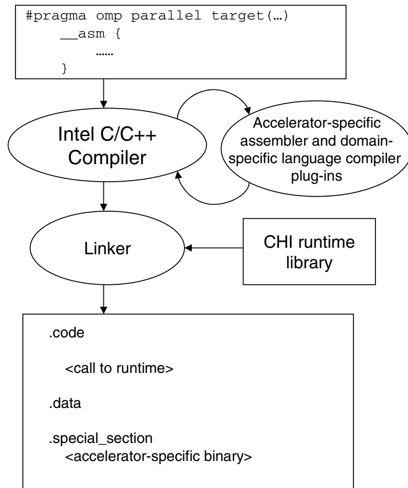

Figure 4. CHI Compilation Flow

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 Section 4.2. As shown in Figure 4, a separate accelerator-specific assembler is dynamically linked with the Intel compiler. 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. With a similar inline compilation mechanism, the CHI compiler also supports integration of a domain-specific high-level language for programming the GMA X3000 hardware.

4.2 OpenMP `Parallel` Pragma Extension

Traditionally, the OpenMP `parallel` directive is used to demarcate a program region for fork-join parallel thread execution. When such a construct is encountered, a number of threads (the thread team) is spawned to execute the dynamic extent of a parallel region. This team of threads, including the main thread that spawned them, participates in the parallel computation. At the conclusion of the parallel region, the main thread waits at an implied barrier until all threads in the thread team complete execution. The main thread then resumes serial execution. The programmer can use additional clauses to specify attributes for the thread team; for example, the `num_threads` clause indicates the number of threads to create.

CHI extends the OpenMP `parallel` pragma. The construct for generating heterogeneous shreds of an accelerator-specific instruction set is outlined in Figure 5(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 IA32 shred encounters an accelerator-specific `parallel` construct with the `target(targetISA)` clause, the IA32 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. By default, the main IA32 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 IA32 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 IA32 sequencer and its exo-sequencers. The CHI runtime is responsible for asynchronously notifying the IA32 sequencer of the eventual completion of all heterogeneous shreds. This concurrency model presents an interesting opportunity for managing parallelism. For example, the programmer may use the heterogeneous shreds to process two thirds of an image while using the main IA32 shred to process the rest of the image in parallel. Section 5.3 further quantifies the benefit of such cooperative heterogeneous multi-shredding.

```
#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 5. CHI Extensions to OpenMP Pragmas

As in standard OpenMP, data communication between the IA32 main shred and the heterogeneous shreds can be specified via data clauses, namely, `shared`, `private`, and `firstprivate`. In the proposed extension to the OpenMP `parallel` directive for heterogeneous shreds, the semantics of these clauses remains identical to their respective meaning for homogeneous symmetric multiprocessors, in both syntax and spirit. By definition, for each variable specified in the `shared` clause, all shreds in a team can access the same memory area. For each variable specified in the `firstprivate` clause, a private copy-constructed variable is created for all shreds with the same value. When the `parallel for` loop is used, the compiler parses the loop construct and, for the variables specified in the `private` clause, each shred context is initialized with different copy-constructed variable values evaluated by each loop iteration. When the team of shreds is launched, each shred executes the same code in the pragma. CHI also introduces a new `descriptor`

clause to allow programmer to optionally associate an accelerator-specific descriptor to a variable enumerated in the `shared` clause. We discuss this further in Section 4.4.

For accelerators highly optimized for thread-level parallelism, such as the GMA X3000, the number of heterogeneous shreds launched may be quite large. For example, as shown in Table 2, a linear filter algorithm for image smoothing, representative of many media filtering operations that are embarrassingly parallel, can launch thousands to tens of thousands of shreds. Each sub-unit (e.g. an 8x8 macroblock) in the input image can be independently processed by a separate parallel shred. Through the use of the `parallel` pragma in CHI, the starting coordinates of the macroblock for each shred to operate on is calculated based on the loop index and passed in through the `private` clause, while the entire source image is shared among all heterogeneous shreds via the `shared` clause. The smoothing constants are then passed to each shred via the `firstprivate` clause. With this OpenMP extension, the programmer can use OpenMP to express the algorithmic thread-level parallelism without having to worry about implementation specific details on how the compiler and runtime translate the parallel construct into shreds and how these shreds are scheduled to run on the exo-sequencers.

4.3 OpenMP Work-Queuing Extension

The fork-join model of the OpenMP `parallel` pragma is an ideal fit for supporting parallel execution of independent threads. 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` [26]. In CHI, we further enhance the compiler and runtime to support inter-shred dependencies among heterogeneous shreds using these pragmas.

The work-queuing model supports the producer-consumer (or pipeline) form of thread level parallelism. It is a flexible mechanism for specifying units of work that are not pre-computed at the start of the work-queuing construct. The construct is implemented by specifying the environment (`taskq`) and the units of work (`task`) separately. A `taskq` pragma creates an empty queue of tasks. The code inside a `taskq` block is executed serially. Any `task` pragma construct encountered while executing a `taskq` block specifies that the enclosed work is associated with the queue. A `taskq` pragma may be nested within either a `taskq` block or a `task` block; in both cases a subordinate queue is formed. The `parallel taskq` construct and the `task` construct for an exo-sequencer are outlined in Figure 5(b) and Figure 5(c).

When an IA32 shred encounters a `taskq` pragma construct with the `target(`*`targetISA`*`)` clause, the IA32 shred calls to the CHI runtime to select a shred as the root shred. The root shred sequentially executes the `while` or `for` loop within the `taskq` construct, and for each `task` pragma encountered the CHI runtime creates a child shred of the root shred and enqueues it into the queue associated with the `taskq`. The `captureprivate` clause creates a private copy-constructed version for each object in *`variable-list`* for the task at the time the task is enqueued.

While the extension to the OpenMP `parallel` pragma is useful for exploiting large numbers of independent shreds through the fork-join model, the `taskq` and `task` work-queuing constructs ensure that the producer-consumer dependency can be honored between GMA X3000 shreds as well. An example that uses such a construct is a deblocking filter for video processing. The recent video coding standards including H.264/AVC follow the block-based hybrid coding approach, in which each picture is represented and processed in block-shaped units. As in block-based transformation and quantization, the block-based processing may induce more distortion in the boundaries of blocks. Deblocking filtering has been used to diminish this kind of block artifact. The deblocking filtering in H.264/AVC is computationally intensive with complicated data access patterns. In order to ensure the quality of the filtering, the deblocking algorithm requires macroblocks to be processed in a particular order; for example, a macroblock will not be processed until its left and upper neighboring macroblocks have been completely processed. Such inter-shred dependency can be easily supported by the work-queuing extension in CHI.

#1	`chi_alloc_desc`(*targetISA,ptr,mode,width,height*)
#2	`chi_free_desc`(*targetISA,desc*)
#3	`chi_modify_desc`(*targetISA,desc,attrib_id,value*)
#4	`chi_set_feature`(*targetISA,feature_id,value*)
#5	`chi_set_feature_pershred`(*targetISA,shr_id,feature_id,value*)

Table 1. CHI APIs for Programming an Exo-sequencer of *targetISA*

4.4 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 the parallel execution on the heterogeneous multi-core target. Table 1 lists the main APIs for the CHI runtime. Like conventional OpenMP runtimes, the CHI runtime layer provides a layer of abstraction that hides the details of managing the exo-sequencers from users of the OpenMP pragmas. Built on the EXO architecture, the runtime layer uses the MISP architectural support for user-level inter-sequencer communication and proxy execution. Like the Shredlib runtime for MISP [11], the CHI runtime is responsible for scheduling and dispatching the heterogeneous shreds for execution and for handling exceptions that may occur on the exo-sequencers.

The compiler translates CHI's OpenMP `parallel` pragma extensions to a series of calls to the runtime layer, which is responsible for appropriately configuring the accelerator hardware for parallel execution. The accelerator-specific assembly block is replaced with a call into a CHI runtime service that is responsible for locating the corresponding accelerator binary code in the fat binary. The CHI runtime service then initiates the parallel execution of the heterogeneous shreds by dispatching shred continuations to the exo-sequencers through the `SIGNAL` instruction. Another critical task for the CHI runtime is to manage data communication between the IA32 and the exo-sequencers, which may have different views on the same shared virtual memory object specified by `shared` data clause, as described in Section 4.2.

General purpose CPU architectures, such as the IA32 family, view the virtual address space as a contiguous one-dimensional linear uniform memory space. For such architectures a memory object (*e.g.*, a global or local variable) in C/C++ can be easily bound to an operand in the inline assembly via a register move or a load instruction. However, domain-optimized accelerators may view memory in a significantly different way than the general purpose CPU [8]. For instance, the GMA X3000 is optimized for 2-D image and video media processing. It accesses virtual memory via *surfaces*, which are two-dimensional blocks of memory. Configuring surface information such as the tiling format is important for achieving the best possible performance in media acceleration code. Even within the same application, one surface can have significantly different attributes from other surfaces.

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. The first three

```
1.  A_desc = chi_alloc_desc(X3000, A, CHI_INPUT, n, 1);
2.  B_desc = chi_alloc_desc(X3000, B, CHI_INPUT, n, 1);
3.  C_desc = chi_alloc_desc(X3000, C, CHI_OUTPUT, n, 1);
4.  #pragma omp parallel target(X3000) shared(A, B, C)
5.    descriptor(A_desc,B_desc,C_desc) private(i) master_nowait
6.  {
7.    for (i=0; i<n/8; i++)
8.      __asm
9.      {
10.       shl.1.w  vr1 = i, 3
11.       ld.8.dw  [vr2..vr9]   = (A, vr1, 0)
12.       ld.8.dw  [vr10..vr17] = (B, vr1, 0)
13.       add.8.dw [vr18..r25]  = [vr2..vr9], [vr10..vr17]
14.       st.8.dw  (C, vr1, 0) = [vr18..vr25]
15.     }
16. }
17. #pragma omp parallel for shared(D,E,F) private(i)
18. {
19.   for (i=0; i<n; i++)
20.     F[i] = D[i] + E[i];
21. }
```

Figure 6. CHI Code Example with GMA X3000 Pseudo-code

APIs listed in Table 1 are used for managing descriptors. API #1 allocates a descriptor by specifying the input/output mode, as well as the width and height of the surface. API #2 deallocates the existing descriptor, and API #3 is a general API to modify the descriptor from its default attributes.

CHI provides two generic runtime APIs for programmers to specify and access specialized features on accelerators. API #4 makes a global change to all exo-sequencers' states, which apply to all heterogeneous shreds created. API #5 changes an exo-sequencer's state on a per shred basis. An application can directly utilize new hardware features simply by making the appropriate call in the source file, without requiring any changes to the compiler.

4.5 Debugging Tools

In the CHI programming environment, the C/C++ compiler, inline assembler and domain-specific language compiler together produce comprehensive source-level debugging information that maps each accelerator-specific instruction to source code. We extend the Intel Debugger (IDB) and the CHI runtime layer to create a debugging environment for the application programmer. The enhanced version of the Intel Debugger is capable of debugging code that is running on the IA32 sequencers as well as the shreds that are running on the exo-sequencers. The debugger extensions consist of two parts. The first part is the introduction of commands to set breakpoints, single-step, and examine state on the GMA X3000 exo-sequencers. The second part comprises the enhancements in the debugger and the CHI run time layer so they can communicate debugging information to one another. This debugger is essential to provide the IA32 look-and-feel in CHI for productive development of heterogeneous multi-shredded code.

4.6 Putting It All Together

Figure 6 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. The example depicts a simple addition of two vectors (A and B) with the results written to a third vector C. The C code (lines 1-16) uses the extended version of the OpenMP `parallel` pragma to perform the vector addition by spreading the computation to `n/8` shreds, with each shred operating on eight elements in the vector using 8-wide SIMD instructions in the GMA X3000 ISA.

In this example, the binding of C variables A, B, C to the inline assembly GMA3000 shred kernel code is done via the `shared` data clause of the `parallel` pragma as specified in line 4. Lines 1-3 illustrate the use of the CHI runtime API #1 to describe the additional information on the surface memory area for the GMA X3000. The surfaces in this example are categorized as either input or output, and described as vector arrays with width of `n` and height of 1. The resulting descriptors for the `shared` variables can be communicated via the `descriptor` data clause of the `parallel` pragma as specified in line 5. Before forking the heterogeneous shreds, the CHI runtime inspects these descriptors and configures the accelerator appropriately. The `private` clause specifies the loop index `i` as an input value for each shred.

The example creates `n/8` GMA X3000 shreds, each executing the `__asm` block. Each shred operates on the same three input vectors, but accesses distinct regions of the surfaces, depending on the per-shred input value `i`. The first line of assembly (line 10) shifts the input value `i` left by 3 bits to calculate the index into the surface and locates the block area to perform computation (line 11-12). After the vector add is executed (line 13), the result is written to the output surface (line 14). The `master_nowait` clause allows the IA32 sequencer to continue executing the traditional OpenMP code (line 17-21) to perform vector add of different arrays without waiting for the pending GMA X3000 shreds to finish, thus creating the parallel execution of both IA32 and GMA X3000 shreds.

5. Performance Evaluation

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 [4]. We select a representative subset of these kernels as benchmarks for performance evaluation. Table 2 summarizes these kernels and inputs. From left to right the columns indicate the kernel name and abbreviation, the input data set, a brief description of the kernel, and finally the total number of shreds spawned per kernel execution. 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 IA32 like SSE. By providing such IA32 look-and-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.

5.1 Performance Speedup on GMA X3000 Exo-sequencers over IA32 Sequencer

Figure 7 shows the speedup achieved over IA32 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

Kernel (Abbreviation)	Data size	Description	# GMA X3000 Shreds
Linear Filter (`LinearFilter`)	640x480 image	Compute output pixel as average of input pixel and eight surrounding pixels	6,480
	2000x2000 image		83,500
Sepia Tone (`SepiaTone`)	640x480 image	Modify RGB values to artificially age image	4,800
	2000x2000 image		62,500
Film Grain Technology (`FGT`)	1024x768 image	Apply artificial film grain filter from H.264 standard	96
Bicubic Scaling (`Bicubic`)	Scale 30 frames 360x240 to 720x480	Scale video using bicubic filter	2,700
Kalman (`Kalman`)	30 frames 512x256	Video noise reduction filter	4,096
	30 frames 2048x1024		65,536
Film Mode Detection (`FMD`)	60 frames 720x480	Detect video cadence so inverse telecine can be applied	1,276
Alpha Blending (`AlphaBlend`)	Blend 64x32 image onto 720x480	Bi-linear scale 64x32 image up to 720x480 and blend with 720x480 image	2,700
De-interlace BOB Avg (`BOB`)	30 frames 720x480	De-interlace video by averaging nearby pixels within a field to compute missing scanlines	2,700
Advanced De-interlacing (`ADVDI`)	30 frames 720x480	Computationally intensive advanced de-interlacing filter with motion detection	2,700
ProcAmp (`ProcAmp`)	30 frames 720x480	Simple linear modification to YUV values for color correction	2,700

Table 2. Media-Processing Kernels

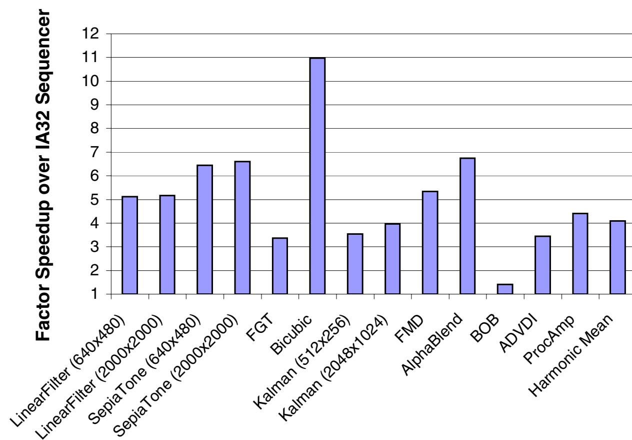

Figure 7. Speedup from Execution on GMA X3000 Exo-sequencers over IA32 Sequencer

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 the 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 IA32 CPU. `LinearFilter`, `ProcAmp` and `SepiaTone` are able to take advantage of such ISA support by aggressively unrolling loops. `AlphaBlending` benefits from the ability to access the texture sampler fixed function unit; in the absence of a texture sampler the IA32 sequencer code has to emulate this behavior in software. `Bicubic` benefits both from the GMA X3000's wide SIMD execution bandwidth and its generous number of general purpose registers (64 to 128).

The lone exception is `BOB`, which achieves a meager, albeit non-trivial, speedup of 1.41X. Compared to the other kernels studied, this kernel is the least computationally intensive. Instead, it is primarily bandwidth-bound, and benefits much less from the greater aggregate execution rate of the GMA X3000 than the other kernels.

5.2 Impact of Data Copying Versus Shared Virtual Address Space

In general, the performance improvement by using an accelerator is determined not only by the accelerator architecture but also by the data communication overhead between the CPU and accelerator. This overhead varies greatly depending on the memory model between the CPU and the accelerator. Figure 7 shows overall performance improvement achieved with a cache coherent shared virtual memory model between the IA32 sequencer and the GMA X3000 exo-sequencers. In the absence of cache coherence or shared memory, the data communication overheads can significantly degrade the speedup achieved by accelerating the computation. In Figure 8 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 IA32 sequencer and the GMA X3000 exo-sequencers. Consequently, data communication between IA32 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 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 IA32 shred and GMA X3000 shreds resemble 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 IA32 sequencer an the GMA X3000 exo-sequencers. Data copying can be avoided in this case as both the IA32 sequencer and GMA

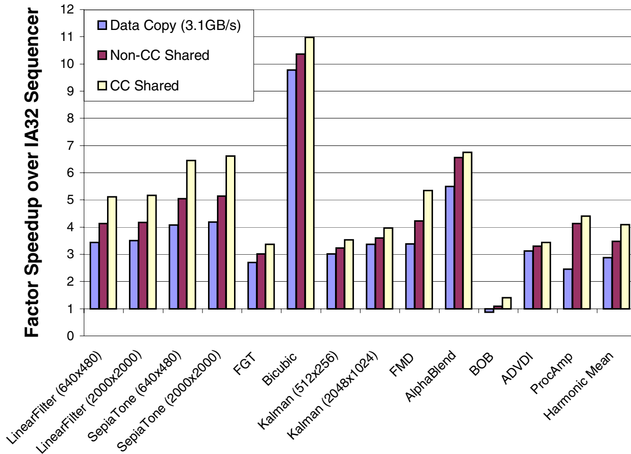

Figure 8. Impact of Shared Virtual Memory

```
1.  n = 800;
2.  GMA_iters = 600;
3.  IN_desc = chi_alloc_desc(X3000, IN, CHI_INPUT, n, 1);
4.  OUT_desc = chi_alloc_desc(X3000, OUT, CHI_OUTPUT, n, 1);
5.  #pragma omp parallel target(X3000) shared(IN, OUT)
6.    descriptor(IN_desc,OUT_desc) private(i) master_nowait
7.  {
8.    for (i=0; i<GMA_iters; i++)
9.      __asm
10.     {
11.       ...
12.     }
13. }
14. #pragma omp parallel for shared(IN, OUT) private(i)
15. {
16.   for (i=GMA_iters; i<n; i++)
17.       ...
18. }
```

Figure 9. Cooperative Execution Code Example which Executes 600 Loop Iterations on GMA X3000 Exo-sequencers and 200 Loop Iterations on the IA32 Sequencer

X3000 exo-sequencers can access the identical physical memory location for the same virtual address. Memory writes performed by the IA32 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 IA32 sequencer and an GMA X3000 exo-sequencer as long as cache flush operations are appropriately invoked. Due to lack of cache coherence, the IA32 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 into the memory.

The third configuration, *CC Shared*, models a cache-coherent shared virtual address space, which is the configuration assumed in Section 5.1. In this model, between the IA32 shred and the GMA X3000 shreds, the data communication becomes much more efficient. Similarly, the synchronization on mutual access to shared data structure is also made much easier to the 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 8 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 IA32 Core 2 Duo, 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 IA32 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 IA32 shred spawns GMA X3000 shreds, it may appear necessary to flush the IA32 sequencer's cache fully before any GMA X3000 shred can be launched. In reality the majority of the cache flush operation on the IA32 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 that data has been flushed back to memory by the IA32 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.

For example, in a system where the cache flush operation has not been optimized and only writes data back to memory at 2GB/s (not shown), executing `LinearFilter` yields a speedup of only 3.15x over IA32 sequencer execution. This occurs if the entire cache flush cost for a 640x480 RGB input must be paid up front before any exo-sequencer shreds are spawned. However, the initial 32 exo-sequencer shreds (which fully populate the GMA X3000 exo-sequencers) access less than 1% of the total input data. By flushing just this necessary data initially, and flushing the remaining data in parallel with execution on the exo-sequencers in the system, performance very close to a cache-coherent shared virtual memory configuration can be achieved without hardware support for cache coherency. This intelligent cache flushing scheme can be carried out by the CHI runtime support, transparent to the programmer.

A shared virtual address space, with or without cache coherence, offers performance benefits by eliminating the cost of copying data between virtual address spaces. More importantly, however, a shared virtual address space allows the IA32 sequencer and the GMA X3000 exo-sequencers to closely cooperate in the execution of a given workload. We discuss this approach further in Section 5.3.

5.3 Cooperative Execution between Heterogeneous Sequencers

The shared virtual address space enabled by EXO allows the IA32 sequencer and GMA X3000 exo-sequencers to simultaneously operate on the same data. Because of this, further speedup is possible by dividing available work between the IA32 sequencer and the GMA X3000 sequencers. Figure 9 illustrates such an example. For each targeted ISA, the programmer provides a separate version of the code to execute a parallel loop, and divides those loop iterations which should execute on each type of sequencer. By using the

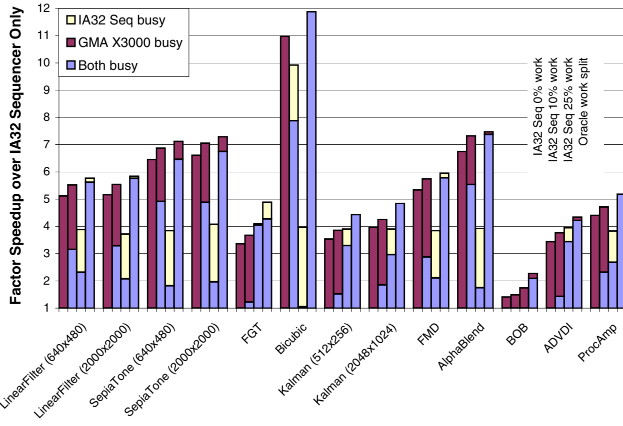

Figure 10. Cooperative Multi-shredding Between IA32 Sequencer and GMA X3000 Exo-sequencers

`master_nowait` clause, cooperative heterogeneous parallel execution can be initiated.

Figure 10 shows the performance improvement when work is divided between the IA32 sequencer and the GMA X3000 exo-sequencers using four different work partitions: (1) 0% of the work is executed by the IA32 sequencer, (2) a static partition where 10% of the work is executed by the IA32 sequencer, (3) a static partition where 25% of the work is executed by the IA32 sequencer and (4) an oracle work partition which optimally distributes the work so that both the IA32 sequencer and GMA X3000 exo-sequencers finish execution as close to the same time as possible. The height of each bar shows the execution time relative to execution on the IA32 sequencer alone, and each bar is divided according to the time when the IA32 sequencer, the GMA X3000 exo-sequencers, or both are busy. This data assumes a cache-coherent shared virtual address space, but by carefully dividing the work among sequencers this technique can also be applicable even if cache coherency is not supported.

For kernels in which the IA32 sequencer performs well, significant speedup is possible, and `BOB`, which shows the best relative IA32 sequencer performance, achieves up to 38% for the oracle scheme. `Bicubic`, on the other hand, sees an improvement of only 8% for the oracle case. While cooperative parallel execution can yield further speedup in all cases, performance is sensitive to work imbalance. In certain cases with poor divisions of labor, *e.g.*, `Bicubic` in partition (3), the performance from cooperative execution is worse than simply executing on the GMA X3000 exo-sequencers. Unfortunately, determining the appropriate work division *a priori* can be challenging, and will only get more complex as the numbers and types of heterogeneous sequencers available continue to grow.

This challenge can be overcome, however, by extending the CHI multi-shredding runtime to support a dynamic work distribution policy. To implement dynamic heterogeneous work scheduling, the programmer can provide a separate version of the code to execute an individual loop iteration for each targeted ISA. At runtime, the multi-shredding runtime creates multiple candidate shred continuations, one for each targeted ISA, for each loop iteration. The multi-shredding runtime then divides the parallel loop iterations among the sequencers in the system. Whenever a sequencer completes its assigned work it requests additional work of the runtime.

Both fork-join and producer-consumer style parallelism can be utilized. If cache coherency is supported, it is easy to extend the traditional shared-memory synchronization primitives, such as locks and mutexes, allowing fine-grained cooperative execution between the heterogeneous sequencers. In systems without cache coherency, mutual exclusion must be ensured in data accesses by the IA32 sequencer and the GMA X3000 exo-sequencers. However, this can be maintained by selectively flushing the necessary data prior to spawning any exo-sequencer shred to consume data produced by the IA32 sequencer. Dynamic work scheduling is an area of ongoing work.

6. Conclusion

As long as Moore's Law continues, the level of on-die device integration will continue to grow. Thus, due to the desire for design flexibility, the need to amortize cost across multiple uses, and the necessity to observe power constraints, future microprocessor designs will bring more heterogeneous building blocks on-die. This level of on-die integration presents challenges as well as opportunities to both microprocessor architecture and the software stack. In particular, this integration will inevitably bring about salient re-architecting of the processor hardware hierarchy, *e.g.*, re-partitioning of power budget to introduce new features, and shedding legacy interfaces to achieve efficiency. Re-partitioning the hardware hierarchy can significantly affect the software stack. On the one hand, there is a need to cope with legacy software ecosystem constraints. On the other hand, a higher degree of on-die hardware integration makes it possible to present traditionally system-level resources as user-level architecture resources, allowing a much tighter coupling between applications and hardware resources. Consequently, hardware resource management can be changed from centralized OS control to application-level control. Such changes can reduce the bloated software stack between application programs and silicon.

In this paper we present the EXO MIMD extension to the IA32 ISA to expose heterogeneous cores as application-level architecture resources and provide shared virtual memory to support the classic multi-shredded programming model for the heterogeneous multi-core. 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 IA32 software development, the CHI programming environment provides an IA32 look-and-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 IA32 look-and-feel of the programming environment has significantly improved productivity over prior device driver-based development environments.

Acknowledgments

We would like to thank John Shen, Rich Hankins, Lisa Pearce, 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, and Dion Rodgers. In addition, we would like to thank Anne Bracy, Ethan Schuchman, Ankur Khandelwal, 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. *ACM Transactions on Graphics*, 23(3):777–786, 2004.

[2] CPU+GPU integration. http://www.google.com/search?hl=en&lr=&rls=GGLG%2CGGLG%2005-47%2CGGLG%3Aen&q=intel+amd+nvidia+ati+cpu+gp%u+integrated+&btnG=Search.

[3] CUDA. http://developer.nvidia.com/object/cuda.html.

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

[5] A. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th international Conference on Parallel Architectures and Compilation Techniques*, 2005.

[6] GLSL OpenGL Shading Language. www.wikipedia.org/wiki/GLSL.

[7] R. Gonzalez. A Software-configurable Processor Architecture. *IEEE Micro*, pages 42–51, Sept-Oct 2006.

[8] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A Memory Model for Scientific Algorithms on Graphics Processor. In *IEEE Supercomputing*, 2006.

[9] GPGPU: General Purpose Computation using Graphics Hardware. www.gpgpu.org.

[10] E. Grochowski and M. Annavaram. Energy per Instruction Trends in Intel Microprocessors. *Technology@Intel Magazine*, March 2006.

[11] 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.

[12] Intel G965 Express Chipset. http://www.intel.com/products/chipsets/g965/prod_brief.pdf.

[13] Intel Santa Rosa Platform. http://www.intel.com/pressroom/archive/releases/20060307corp_b.htm.

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

[15] *Intels Next Generation Integrated Graphics Architecture Intel Graphics Media Accelerator X3000 and 3000*. Intel Corporation, 2006.

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

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

[18] 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.

[19] 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*, (3):896–907, 2003.

[20] M. McCool and S. Toit. *Metaprogramming GPUs with Sh*. A K Peters, 2004.

[21] M. McCool, K. Wadleigh, B. Henderson, and H. Y. Lin. Performance Evaluation of GPUs using the RapidMind Development Platform. In *Proceedings of the 20th International Conference on Supercomputing*, 2006.

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

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

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

[25] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallelism in OpenMP. In *First European Workshop on OpenMP*, September 1999.

[26] 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 *Proceedings of the 4th European Workshop on OpenMP*, 2002.

[27] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.

[28] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Computational Complexity*, 2002.

[29] 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.

[30] 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.

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

[32] D. Zhang, Z. Li, H. Song, and L. Liu. A Programming Model for an Embedded Media Processing Architecture. In *Embedded Computer Systems: Architecture, Modeling, and Simulation*, 2005.