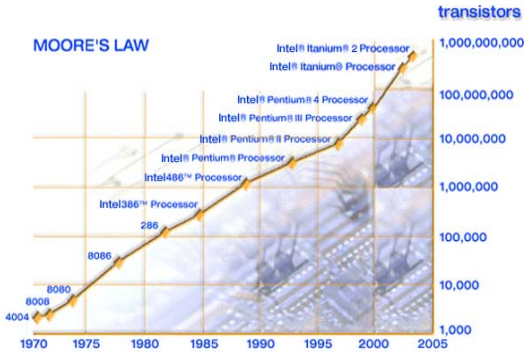
EXOCHI

Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System

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

Presenter: Wes Toland



Moore's Law is the observation that the number of transistors that can be inexpensively placed on an integrated circuit is increasing at an exponential rate (doubling about every two years)

Motivation

- Moore's law trend is expected to continue for about 10 years *but*…
- Increasing the amount of transistors per die creates *power consumption* and *cooling* issues
- Several techniques can be used to improve performance without increasing the number of transistors:
  - Enhanced pipelines
  - Larger caches
  - **Many execution cores per die**
- This paper proposes a customized heterogeneous multi-core architecture and programming environment

Outline

- Overview
- Challenges
- Related Work
  - Heterogeneous Programming Models
  - Heterogeneous Execution Models
- What is EXOCHI?
  - EXO Architecture
  - CHI Programming Environment
- Evaluation
- Future Work
- Conclusion

Overview

- Multi-core architectures are *homogeneous* or *heterogeneous*
  - Homogeneous implies all cores are identical
  - Heterogeneous is a collection of cores with different ISAs
- Many heterogeneous multi-core architectures use accelerators to increase performance and efficiency
  - Graphics Processing Units (GPUs)
  - FPGAs

Challenges (1/3)

- Making scalable multi-core architectures is not possible without low energy-per-instruction (EPI) cores
- Custom-cores, such as GPUs, tend to have lower EPIs than general-purpose cores
  - Custom-cores usually have a limited set of functionality

Challenges (2/3)

- Functionality provided by most custom architectures is usually not robust enough to handle most applications
- Some heterogeneous designs use a set of custom cores and general purpose CPUs
  - This type of platform is not limited to a specific application domain

Challenges (3/3)

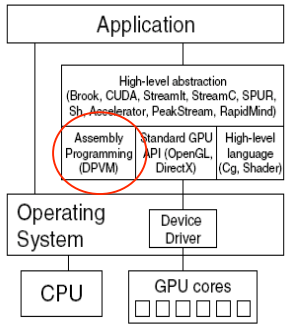
- Heterogeneous multi-core architectures are difficult to program
- The IA32 model (for CPUs) is the most popular model
- Heterogeneous models attempt to mimic the IA32-based model
  - Details of architecture are hidden from user
  - This can often result in less opportunity for low-level optimizations

Heterogeneous Programming Models

- GPGPU programming models have been designed for GPU-based heterogeneous architectures
  - Several layers of abstraction
    - Lowest level: Programmer can deal directly with the assembly of the ISA
    - Highest level: Run time systems hide system architecture details from the programmer

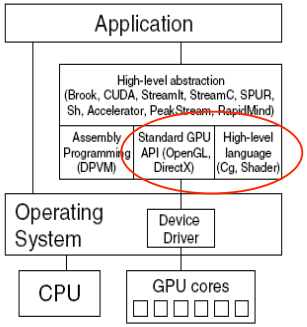
Assembly-level programming

- Low level of abstraction
  - Data Parallel Virtual Machine (DPVM)
  - Program GPU with native ISA and device drivers
  - Programmer explicitly manages GPU devices
  - Communication method: Explicit copying via device drivers



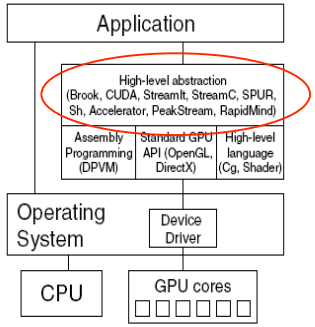
GPU APIs & Higher Level Languages

- Intermediate layer of abstraction
  - Program GPU with domain-specific APIs or higher level languages
  - Makes ISA and GPU devices transparent to programmer
  - Drawbacks
    - These APIs are mostly domain-specific programming models for graphics processing
    - Some algorithms and data structures do not naturally fit these models



Streaming Programming Models

- High level of abstraction
  - GPU is abstracted as stream processor
  - Programmers can easily parallelize code without managing any underlying resources
  - High-level data structures are supports (C/C++)
  - Benefit: Higher productivity
  - Drawback: Harder to get optima performance from GPUs



Domain-specific Virtual Machines

- Highest level of abstraction
  - Typically exists as libraries that manage the computations across the different ISAs for the user
  - Benefit: Takes advantage of GPU acceleration
  - Drawback: Advanced ISA-specific optimizations are not possible
  - Several domain-specific virtual machines have been successfully developed:
    - Stanford Stream Virtual Machine
    - RapidMind Streaming Execution Manager
    - Microsoft Accelerator for data parallelism exploitation

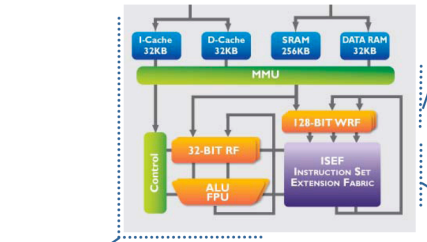
Heterogeneous Execution Models

- 2 main categories:
  - ISA-based tightly-coupled approach
  - Device driver-based loosely-couple execution model
- EXOCHI is a unique architecture and doesn't fit perfectly into either of these categories

ISA-based Tightly-coupled Approach (1/2)

- Example: Software-Configurable Processor (SCP) Architecture
  - Custom ISA extensions virtually represent the operation implemented on hardware accelerators
  - Master CPU decodes these custom instructions and dispatches the appropriate co-processor instruction
  - Master CPU blocks while these co-processor instructions execute

ISA-based Tightly-coupled Approach (2/2)



Wide Register File
- 32 128-bit entries

Load/store unit
- 128-bit load/store
- Auto increment/decrement
- Immediate, indirect, circular
- Variable-byte load/store
- Variable-bit load/store

ISEF
- 3 inputs and 2 outputs
- Pipelined, interlocked
- 32 16-bit MACs and 256 ALUs
- Bit-sliced for arbitrary bit-width
- Allow control logic
- Allow internal processor state

RISC Processor
- Tensilica – Xtensa V
- 32 KB I & D Cache
- On-Chip Memory, MMU
- 24 Channels of DMA, FPU

Strech S5000 SCP Architecture

http://www.stanford.edu/class/ee380/Abstracts/SoftwareConfigurableProcessor.pdf

Device driver-based loosely-coupled execution model

- GPGPU infrastructures
  - Main CPU resources are managed by the operating system
  - GPU resources are separately managed by commercial device drivers
  - Applications and device drivers have separate address spaces
    - Synchronization is accomplished with explicit data copying via commercial device driver APIs

Heterogeneous Execution Models

- EXOCHI was developed to overcome the shortcomings of the 2 main heterogeneous execution models
- ISA-based Tightly-coupled Approach
  - Master thread has to wait while co-processor code blocks execute
  - Most implementations do not allow for inter-accelerator communication
- Device driver-based loosely-coupled execution model
  - Data copying via device drivers is very inefficient
- EXOCHI will allow for a more efficient, and more parallel environment

What is EXOCHI?

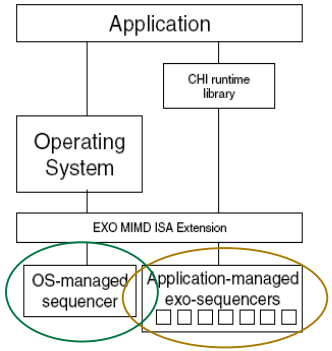
- Exoskeleton Sequencer (EXO)
  - Heterogeneous accelerators are represented as ISA-based MIMD resources
  - Accelerators are tightly coupled with the general purpose CPUs via shared virtual memory
- C for Heterogeneous Integration (CHI)
  - Fully integrated programming environment for the EXO architecture
  - Supports accelerator-specific inline assembly
  - Supports domain-specific languages
  - Extends OpenMP pragmas

EXOCHI vs. existing models

- EXOCHI does not fit into either execution models
- Differences from tightly-coupled approaches:
  - Independent sequencing
  - Concurrent execution of multiple instruction streams on multiple sequencers within single OS thread context
- Differences from loosely-coupled, driver-based approaches:
  - The heterogeneous sequencers are available to programmers
  - There is a virtual address space between sequencers.

EXOCHI Overview

- EXO architecture
  - OS-managed sequencer
    - Intel Core 2 DUO
    - EFI: 10 nJ
  - Application-managed exo-sequencers
    - Intel Graphics Media Accelerator 3000
    - EFI: 0.3 nJ



EXO Architecture

- EXO extends the (Multiple Instruction Stream Processing) MISP architecture
- EXO support address translation remapping (ATR)
- EXO support collaborative exception handling (CEH)

MISP

- One of the first architectures to introduce a sequencer as a type of architectural resource
  - Can easily be extended to handle different sequencer ISAs
- User-level inter-sequencer signaling API
- Asynchronous control transfer
- Applications can manage user-level threads without OS intervention
- Supports cache-coherent shared memory model

MISP exoskeleton

- "exoskeleton" wrapper
  - Extended inter-sequencer signaling mechanism to support non-IA32 sequencers
- Exo-sequencers
  - non-IA32 accelerators that use the "exoskeleton" wrapper
- Exoskeleton integrated with the OS-managed IA32 sequencer
  - inter-sequencer user-level interrupts makes this possible
  - IA32 sequencer uses MISP **SIGNAL** instruction to spawn non-IA32 "shreds" to run on exo-sequencers

Address Translation Remapping

- Handling exo-sequencer page faults would be difficult to handle in hardware
  - Exo-sequencers cannot directly access IA32 page table when experiences page fault
  - Page formats typically different
- ATR handles page faults in exo-sequencers
  - Upon page fault, exo-sequencer suspends execution and requests **proxy execution** from IA32 sequencer
  - **Proxy execution** services the TLB miss or page fault

Proxy Execution in MISP

- MISP implementation
  - Programmer defines trigger set of user-level fault exceptions that occur on application-managed sequencers (AMS)
  - Trigger conditions cause fault exception to be relayed from the AMS to the OS-managed sequencers (OMS)
  - OMS then performs asynchronous control transfer and suspends execution
  - Then the proxy handler executes any proxy operations for the AMS

Proxy Execution in EXOCHI

EXOCHI Implementation

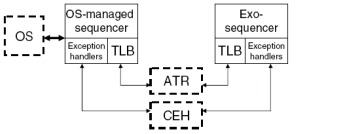


Figure 2. ATR and CEH between Heterogeneous Sequencers

- Proxy execution differs in EXOCHI
  - Once IA32 sequencer handles page fault, ATR will convert the page table entry into the exo-sequencer's page format before the entry is added to the TLB
  - Exo-sequencer's TLB points to the same physical page as the IA32's TLB
    - Allows for direct data access

Address Translation Remapping: Limitations

- ATR does not guarantee cache coherence between sequencer and accelerator
- Critical sections required for data protection
- Cache flushes required when:
  - IA32 shred releases shared data structure to exo-sequencer
  - Exo-sequencer finishes computation and wants to return execution to IA32 sequencer
  - Flushes commit any dirty blocks to memory
- Full cache coherence would be desirable
  - Critical sections can be avoided

Collaborative Exception Handling: Motivation

- Some exo-sequencer exceptions require OS action
- Standard MISP uses proxy execution to replay exceptions encountered by application-managed sequencers on OS-managed sequencer
- In EXOCHI, faults on non-IA32 exo-sequencers cannot be replayed on OS-managed sequencer
  - Different ISA
  - Specific data types may not be supported
  - Specific exception handling may not be supported

Collaborative Exception Handling: Solution

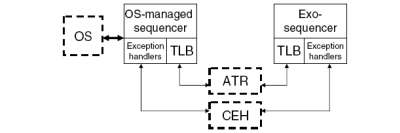


Figure 2. ATR and CEH between Heterogeneous Sequencers

- Hybrid solution (Hardware + Software)
- Exo-sequencer signals IA32 sequencer upon exception
- IA32 sequencer acts as a proxy for the exo-sequencer and handles the exception
- CEH ensures the result is updated in the exo-sequencer before resuming execution

EXO Prototype (1/2)

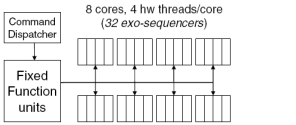


Figure 3. Intel Graphics Media Accelerator X3000

- OS-managed sequencer
  - Intel Core 2 Duo Processor (IA32)
- Exo-sequencer
  - Intel 965G Express Chipset
    - Integrated Intel Graphics Media Accelerator X3000
    - 8 programmable GPGPU cores, each with 4 thread contexts

EXO Prototype (2/2)

- Thread management
  - GMA X3000 shreds can be spawned by IA32 or GMA X3000 shred
- GMA X3000 shreds are scheduled in work queue in shared virtual memory (similar to POSIX)
- Exo-sequencers can write into each other's register files
- GMA X3000 ISA is optimized for data- and thread-level parallelism
  - Each exo-sequencer supports SIMD operations on up to 16 elements in parallel

CHI Programming Environment

- 3 main goals:
  - Provide a mechanism to inline accelerator- or domain-specific code in high-level code
  - Provide tools that allow programmers to easily write programs that achieve shred-level parallelism in accelerator-specific code
    - Fork-join model
    - Producer-consumer model
  - Develop a mechanism to manage memory
    - Input memory regions
    - Output memory regions
    - Live-in values for accelerator-specific code

CHI Extensions

- Inline Accelerator Assembly support
- OpenMP `parallel` Pragma extension
- OpenMP Work-Queuing Extension
- CHI Runtime Support
- Debugging Tools

CHI Framework

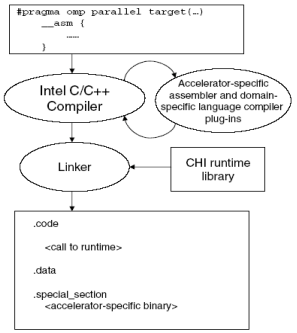


Figure 4. CHI Compilation Flow

Inline Accelerator Assembly Support

- Use C/C++ mechanism to inline assembly code blocks to generate exo-sequencer inline assembly routines
- OpenMP `target` pragma used to specify target ISA of assembly code blocks
- Assembly blocks are compiled into an accelerator-specific binary by accelerator-specific assembler
- All compiled assembly blocks are placed into a single “fat binary”

OpenMP Pragma Extensions: Fork-join model

- `parallel` construct
  - Fork-join thread execution
    - Set of threads are spawned to execute parallel region with the CHI runtime system encounters this construct
    - There is an implied barrier at end of parallel region
  - CHI runtime is responsible for notifying the IA32 sequencer when all exo-sequencers have completed execution

OpenMP Pragma Extensions: Fork-join model

```
#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

- `target` clause specifies which accelerator ISA to use in region
- `master_nowait` clause allows the main IA32 thread to continue execution past the shared region.
  - This can be used to achieve concurrent execution on both sequencer and accelerator

OpenMP Pragma Extensions: Fork-join model


```
#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

- Data clauses (for inter-shred communication):
  - `shared`: All shreds can access same memory area
  - `firstprivate`: a private copy-constructed variable is created for all shreds with the same value
  - `private`: the shred's context is initialized with different copy constructed variable values for each loop iteration

OpenMP Work-Queuing Extension

- Motivation
  - OpenMP fork-join model is for fork-join model only, irregular parallelism cannot be achieved
- Intel C++ compiler supports irregular parallelism through OpenMP pragmas `taskq` and `task`
  - This is used to achieve producer-consumer thread-level parallelism

OpenMP Work-Queuing Extension

```
#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`

- `taskq` pragma is the queue specification mechanism
- It creates an empty queue of tasks
- `taskq` blocks of code are executed sequentially

- Any `task` pragma that is encountered within a `taskq` block will be associated with that queue

```
#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)`

OpenMP Work-Queuing Extension

- When a `taskq` pragma with `target` clause is encountered, the IA32 shred notifies the CHI runtime to select a root shred
- The root shred sequentially executes code within the `taskq` construct
- The CHI runtime creates a child shred of the root for every `task` pragma encountered
  - Child is enqueued into the queue associated with the `taskq` pragma

CHI Runtime Support: Overview

- Acts as a shred scheduler
  - OpenMP directives translated into primitives
  - These primitives create and maintain shreds
- Handles exo-sequencer exceptions
- Runtime layer hides exo-sequencer management from users
  - MISP user-level inter-sequencer support
  - MISP proxy execution

CHI Runtime Support: Memory Management Support (1/2)

- General-purpose processors assume a 1-dimensional contiguous memory space
- Domain-optimized processors often view memory differently
- CHI runtime APIs are provided to send accelerator-specific information through descriptors
- Accelerator uses descriptors to interpret shared variable attributes

CHI Runtime Support: Memory Management Support (2/2)

#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*

- `chi_alloc_desc`: allocates descriptor by specifying the input/output mode, as well as the width and height of the surface
- `chi_free_desc`: de-allocates memory descriptor
- `chi_modify_desc`: modify descriptor's attributes
- `chi_set_feature`: make global change to all exo-sequencers' states
- `chi_set_feature_pershred`: changes exo-sequencer's state on a per shred basis

CHI Debugging Tools

- Extended Intel Debugger (IDB) on top of CHI runtime layer
- Programmers can debug code running on IA32 sequencers as well as on exo-sequencers
- Two main implementation steps:
  - Implement commands to set break-points, single-step, and examine other components of exo-sequencers
  - Enhance IDB and the CHI runtime layer to allow for exchange of debugging information

CHI Programming Model (1/4)

- Addition of 2 vectors, A and B, result stored in vector C
- Lines 1-16 use the OpenMP parallel pragma to perform vector addition in parallel
- The computation is spread to n/8 shreds

```
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

CHI Programming Model (2/4)

- Lines 1-3: CHI runtime API #1
- Describes the surface memory area for GMA X3000
- Each vector is tagged as input or output
- Vector arrays have width n, height 1

```
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

CHI Programming Model (3/4)

- Line 4: Binds vectors A,B, and C to the inline assembly of the GMA X3000
- `shared` data clause of the `parallel` pragma

```
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

CHI Programming Model (4/4)

- Line 5: Shared variables descriptors are specified with the `descriptor` data clause of the `parallel` pragma
- `private` clause specifies index i, the input value for each shred
- `master_nowait` clause allows IA32 sequencer to continue execution after all the threads are spawned

```
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

Evaluation Overview

- Several benchmark kernels were hand chosen based on potential task- and data-level parallelizations
- Benchmark kernels were optimized for GMA X3000
  - Wide SIMD instructions
  - Predication support
  - Large register files (64-128 vector registers per exo-sequencer)
- First experiment assumes cache coherent shared virtual memory
- Speedup was measured between IA32 and IA32 with accelerator execution

Benchmark Kernels

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

The last column is the total number of shreds spawned per kernel execution

Benchmark Speedup

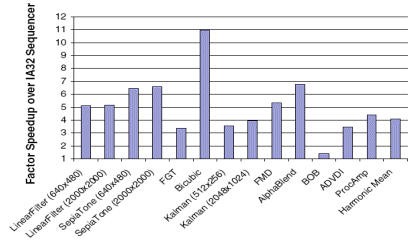


Figure 7. Speedup from Execution on GMA X3000 Exo-sequencers over IA32 Sequencer

- Speedup
  - LOW: BOB -1.41x
  - HIGH: Bicubic -10.97x

Benchmark Speedup Analysis

- Speedup was accomplished due to:
  - High availability of shred-level parallelism (less context-switching)
  - Good thread-level parallelism
    - GMA X3000 has 32 hardware threads that can read/write multiple data streams
  - Good data-level parallelism
    - Wide SIMD operations
  - CHI Runtime allows user to manage shred scheduling manually
    - Programmers can organize shreds in work queue to exploit spatial or temporal locality

Well-suited Kernels for EXOCHI

- LinearFilter, ProcAmp, and SepiaTone
  - ISA support for optimizations such as loop unrolling improved performance
- AlphaBending
  - Ability to access a texture sampler fixed function unit in the GMA X3000 chip improved performance
- Bicubic
  - GMA X3000's wide SIMD execution bandwith and the large number of general purpose registers improved performance
- BOB
  - Experienced lowest speed-up because it is the least computationally intensive (it is more IO-bound)

Impact of data copying (1/2)

- The first experiment used cache-coherent shared virtual memory
  - Replacing this communication with data copying can seriously hinder performance
  - Data communication is on of the most important aspects of multi-core architectures
- Another experiment was performed to compare 3 different memory models:
  - Data Copy
    - No cache coherence of shared virtual memory
    - Data copying is primary method of communication
  - Non-CC Shared
    - Shared virtual address space, no cache coherence
    - Critical sections used instead of data copying
  - CC-Shared
    - Cache-coherent shared virtual memory

Impact of data copying (2/2)

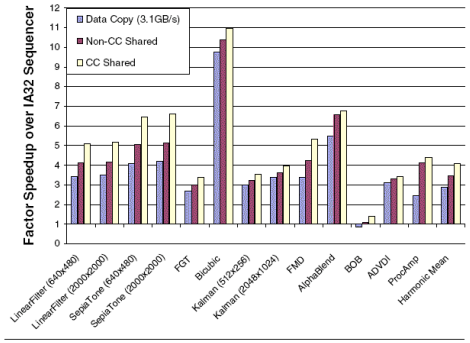


Figure 8. Impact of Shared Virtual Memory

Cooperative Execution Between Heterogeneous Sequencers (1/3)

- IA32 sequencer and GMA X3000 exo-sequencers can operate on same data simultaneously via `master_nowait` clause

```
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

Cooperative Execution Between Heterogeneous Sequencers (2/3)

- Several tests were performed to measure performance impact of sharing work between sequencer and accelerator
- Cache-coherent shared virtual address space
- 4 different work partitions:
  - 0% work done by IA32
  - Static partition where 10% work done by IA32
  - Static partition where 25% work done by IA32
  - Oracle work partition that efficiently distributes work such that the IA32 and GMA X3000 finish tasks as close to each other as possible

Cooperative Execution Between Heterogeneous Sequencers (3/3)

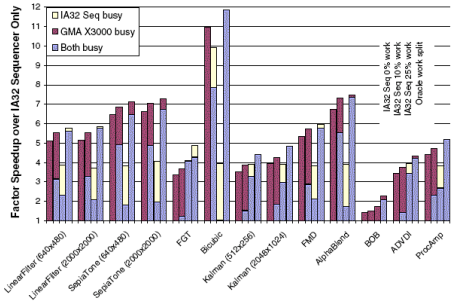


Figure 10. Cooperative Multi-shredding Between IA32 Sequencer and GMA X3000 Exo-sequencers

Future Work

- This team believes they have a solution for work imbalance
  - Extend the CHI runtime to support dynamic work distribution
  - Programmer has to provide a separate version of code to execute one iteration of a loop for each target ISA
  - Runtime system creates candidates for each loop iteration for each target ISA
  - These candidates are used to efficiently divide parallel-loops over available sequencers

Conclusion

- The EXO architecture is on the cutting edge of heterogeneous models.
- The CHI environment is the highlight of this work.
  - The OpenMP extensions on top of the MISP framework make for a nice, intuitive programming environment.
- Speed-up was impressive for computationally-intensive parallelizable applications
- Dynamic work distribution would make this architecture apply to almost any industry.