High-Performance Execution of Multithreaded Workloads on CMPs

M. Aater Suleman
Advisor: Yale Patt

HPS Research Group
The University of Texas at Austin
How do we use the transistors?

• More transistors $\Rightarrow$ Higher performance core
  – Performance increases without programmer effort
  – Larger cores are complex and consume more power

• More transistors $\Rightarrow$ Bigger cache
  – Assist the core by reducing memory accesses
  – Easier to design and consume less power
  – Pentium M: 50M out of the 77M were cache

But, do CMPs improve performance?

• More transistors $\Rightarrow$ More cores
  – Chip Multiprocessors (CMPs)
  – Less complex
  – Run at lower frequency (Power $\alpha$ frequency$^3$)
Multithreading

But, can we do this for all applications?

To leverage CMPs, applications must be split into *threads*.
Easy-to-parallelize Kernels

GrayscaleToMonochrome (picture)

```plaintext
foreach (OldPixel in picture)
  if (OldPixel > Threshold)
    NewPixel = 1
  else
    NewPixel = 0
```

Kernel from ImageMagick
Serial Kernels

Old pixels:

New pixels:

Smooth(Picture)

for i = 1 to N
Pixel[i] = (Pixel[i-1] + Pixel[i])/2
Amdahl’s Law

As the number of cores increase, even a small serial part can have significant impact on overall performance

Future CMPs must improve performance of both parallel and serial parts
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)

• Summary
Current CMP Architectures
Current CMP Architectures

<table>
<thead>
<tr>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
</tbody>
</table>

“Niagara” Approach

- Tile many small cores
- Sun Niagara Processor
- High throughput on the parallel part
- Low performance on the serial part
Current CMP Architectures

<table>
<thead>
<tr>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
</tbody>
</table>

“Niagara” Approach
Current CMP Architectures

- **“Niagara” Approach**
  - Tile a few large cores
  - IBM Power 5, AMD Barcelona, Intel Core2Quad
  - High performance on the serial part
  - Low throughput on the parallel part

- **“Tile-Large” Approach**
## Current CMP Architectures

### "Niagara" Approach

<table>
<thead>
<tr>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
</tbody>
</table>

### "Tile-Large" Approach

<table>
<thead>
<tr>
<th>Large core</th>
<th>Large core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large core</td>
<td>Large core</td>
</tr>
</tbody>
</table>
The Asymmetric Chip Multiprocessor (ACMP)

“Niagara” Approach

- Provide one large core and many small cores
- Accelerate serial part using the large core
- Execute parallel part on small cores for high throughput

“Tile-Large” Approach

ACMP Approach
### The Asymmetric Chip Multiprocessor (ACMP)

<table>
<thead>
<tr>
<th>“Niagara” Approach</th>
<th>“Tile-Large” Approach</th>
<th>ACMP Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Large core</td>
<td>Large core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Large core</td>
<td>Large core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Large core</td>
<td>Large core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Large core</td>
<td>Large core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Large core</td>
<td>Large core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Large core</td>
<td>Large core</td>
</tr>
</tbody>
</table>
The Asymmetric Chip Multiprocessor (ACMP)

<table>
<thead>
<tr>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
</tbody>
</table>

- **Large core**
- **Large core**

<table>
<thead>
<tr>
<th>Large core</th>
<th>Large core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large core</td>
<td>Large core</td>
</tr>
</tbody>
</table>

- **“Niagara” Approach**
- **“Tile-Large” Approach**
- **ACMP Approach**

**Analytical experiment details**
- One large core replaces four small cores
- Large core provides 2x performance
Performance vs. Parallelism

At medium parallelism, ACMP wins and outperforms Niagara.

At high parallelism, Niagara outperforms ACMP.

Niagara beats ACMP at 97% parallelism.

Both ACMP and Tile-Large outperform Niagara.
## Throughput of ACMP vs. Niagara

<table>
<thead>
<tr>
<th>Large core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
<th>Niagara-like core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
<tr>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
<td>Niagara-like core</td>
</tr>
</tbody>
</table>
ACMP Scheduling

ACMP Approach
Data Transfers in ACMP

• Data is transferred if the serial part requires the data generated by the parallel part or vice-versa

• ACMP
  – Data is transferred from all small cores

• Niagara/Tile-Large
  – Data is transferred from all but one core

• Number of data transfers increases by only 3.8%
Experimental Methodology

• Configurations:
  – Niagara: 16 small cores
  – Tile-Large: 4 large cores
  – ACMP: 1 large core, 12 small cores

• Simulated existing multithreaded applications without modification

• Simulation parameters:
  – x86 cycle accurate processor simulator
  – Large core: 2GHz, out-of-order, 128-entry window, 4-wide issue, 12-stage pipeline
  – Small core: 2GHz, in-order, 2-wide, 5-stage pipeline
  – Private 32 KB L1, private 256KB L2
  – On-chip interconnect: Bi-directional ring
Performance Results

![Performance Results Diagram]

- **Tile-Large**
- **ACMP**

The diagram shows the speedup over Niagara for different parallelism levels: Low, Medium, and High. Each bar represents the speedup for various applications such as mcf, is_nasp, fft_splash, and others.
Impact of ACMP on Programmer Effort

• ACMP makes performance less dependent on length of the serial part

• Programmers parallelize the easy-to-parallelize kernels

• Hardware accelerates the difficult-to-parallelize serial part

• Higher performance can be achieved with less effort
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)

• Summary
How Many Threads?

- Some applications:
  - As many threads as the number of cores
- Other applications:
  - Performance saturates
  - Fewer threads than cores

The number of threads must be chosen carefully.
Two Important Limitations

• Contention for shared data
  – Data synchronization: Critical section

• Contention for shared resources
  – Off-chip bus
Contention for Critical Section

All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
All work and no play makes Jack a dull boy.
Contention for Critical Section

Critical Section:
Add local histogram to global histogram

GetPageHistogram(Page *P)

UpdateLocalHistogram(Fraction of Page)

Critical Section:
Add local histogram to global histogram

Barrier

Parallel Part

Serial Part

Kernel from PageMine
Contention for Critical Section

- **Time outside Critical Section (CS)**
- **Time inside CS**

<table>
<thead>
<tr>
<th>N</th>
<th>Time outside CS</th>
<th>Time inside CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LEGEND**
- Parallel Portion
- Critical Section
- Waiting for Critical Section
Two Important Limitations

• Contention for shared data
  – Data-synchronization: Critical section

• Contention for shared resources
  – Off-chip bus
Off-Chip Bandwidth
Contestion for Off-chip Bus

EuclideanDistance (Point A)

\[
\text{for } i = 1 \text{ to } \text{num\_dimensions} \\
\text{sum} = \text{sum} + A[i] \times A[i]
\]
Contention for Off-chip Bus

N = 1
N = 2
N = 4
N = 8

N=4 and N=8 take same time to execute

LEGEND
- Parallel Part
- Bus Access
- Waiting for Bus
Who Chooses Number of Threads?

• Programmer
  – No! Not for general-purpose workloads
    Large variation in input data and machines

• User
  – No! I do not want Windows media player to ask me
    the number of threads

• Set equal to the number of cores
  – Assumption:
    More threads → More performance

Goal: A run-time mechanism to estimate the best number of threads
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)
    • Synchronization-Aware Threading (SAT)
    • Bandwidth-Aware Threading (BAT)
    • Combining SAT and BAT (SAT+BAT)

• Summary
Feedback-Driven Threading (FDT)

Conventional Multithreading

N = K

Feedback-Driven Threading

N = No. of threads
K = No. of cores

Train to sample application behavior
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)
    • Synchronization-Aware Threading (SAT)
    • Bandwidth-Aware Threading (BAT)
    • Combining SAT and BAT (SAT+BAT)

• Summary
Synchronization-Aware Threading (SAT)

\[ T_N = \frac{\text{Time outside C.S.}}{N} + N \times \text{Time inside C.S.} \]

\[ N_{CS} = \sqrt{\frac{\text{Time outside C.S.}}{\text{Time inside C.S.}}} \]
Implementing SAT using FDT

• Train
  – Measure the time inside and outside the critical section using cycle counter

• Compute $N_{CS} = \sqrt{\frac{\text{Time outside C.S.}}{\text{Time inside C.S.}}}$

• Execute
Machine Configuration

- CMP: 32 in-order cores (2-wide, 5-stage deep)
- Caches: L1: 8-KB, L2: 64KB. Shared L3: 8MB
- Off-chip bus: 64-bit wide, 4x slower than cores
- Memory: 200 cycle minimum latency
Results of SAT

SAT decreases execution time and saves power
Adaptation of SAT to Input Data

- Time inside and outside the critical section depends on the input to program
- For PageMine, the best number of threads changes with the page size
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)
    • Synchronization-Aware Threading (SAT)
    • Bandwidth-Aware Threading (BAT)
    • Combining SAT and BAT (SAT+BAT)

• Summary
Bandwidth-Aware Threading (BAT)

N = 1
N = 2
N = 4
N = 8

N = 4 and N = 8 take same time to execute

LEGEND
- Parallel Part
- Bus Access
- Waiting for Bus

\[ N_{BW} = \frac{\text{Total Bandwidth}}{\text{Bandwidth used by a single thread}} \]
Implementation BAT using FDT

• Train
  – Measure bandwidth utilization using performance counters

• Compute \( N_{BW} = \frac{\text{Total Bandwidth}}{\text{Bandwidth used by a single thread}} \)

• Execute
Results of BAT

BAT saves power without increasing execution time
Adaptation of BAT to System Configuration

- The best number of threads is a function of off-chip bandwidth.
- BAT correctly predicts the best number of threads for systems with different bandwidth.

The graph shows the normalized execution time (y-axis) against the number of threads (x-axis) for different bandwidth rates. The graph includes lines for \(\frac{1}{2}\)x bandwidth, 2x bandwidth, and BAT, demonstrating how BAT accurately predicts performance across various bandwidth conditions.
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)
    • Synchronization-Aware Threading (SAT)
    • Bandwidth-Aware Threading (BAT)
    • Combining SAT and BAT (SAT+BAT)

• Summary
Combining SAT and BAT

- Train
  - Train for both SAT and BAT

- Compute
  \[ N_{SAT+BAT} = \text{MIN} (N_{CS}, N_{BW}, \text{Num. cores}) \]

- Execute
Results of SAT+BAT

Fewer threads $\rightarrow$ fewer cache misses

(SAT+BAT) reduces power and execution time

On average, (SAT+BAT) reduces the execution time by 17% and power by 59%
Comparison with Static-Best

Simulate all possible number of threads and choose the best

Two kernels: First needs 12 threads, second needs 32. Static-Best uses 32 for both.
Outline

• Background

• Speeding up serial part
  – Asymmetric Chip Multiprocessor (ACMP)

• Speeding up parallel part
  – Feedback-Driven Threading (FDT)
    • Synchronization-Aware Threading (SAT)
    • Bandwidth-Aware Threading (BAT)
    • Combining SAT and BAT (SAT+BAT)

• Summary
Summary

• CMPs have increased the importance of multithreading

• Performance of both serial and parallel parts is important

• Asymmetric Chip Multiprocessor (ACMP)
  – Accelerates the serial portion using a high-performance core
  – Provides high throughput on the parallel portion using multiple small cores

• Feedback-Driven Threading (FDT)
  – Estimates best number of threads at run-time
  – Adapts to input sets and machine configurations
  – Does not require programmer/user intervention
• Thank You