Performance Monitoring And Optimizations
Review: Keys to Parallel Performance

- **Coverage** or extent of parallelism in algorithm
  - Amdahl’s Law

- **Granularity** of partitioning among processors
  - Communication cost and load balancing

- **Locality** of computation and communication
  - Communication between processors or between processors and their memories
Overlapping Communication with Computation

- Get Data
- Compute

CPU is idle
Memory is idle

synchronization point
Limits in Pipelining Communication

- Computation to communication ratio limits performance gains from pipelining

- Where else to look for performance?
Artifactual Communication

- Determined by program implementation and interactions with the architecture

- Examples:
  - Poor distribution of data across distributed memories
  - Unnecessarily fetching data that is not used
  - Redundant data fetches
Lessons From Uniprocessors

- In uniprocessors, CPU communicates with memory.

- Loads and stores are to uniprocessors as "get" and "put" are to distributed memory multiprocessors.

- How is communication overlap enhanced in uniprocessors?
  - Spatial locality
  - Temporal locality
Spatial Locality

- CPU asks for data at address 1000
- Memory sends data at address 1000 … 1064
  - Amount of data sent depends on architecture parameters such as the cache block size
- Works well if CPU actually ends up using data from 1001, 1002, …, 1064
- Otherwise wasted bandwidth and cache capacity
Temporal Locality

- Main memory access is expensive
- Memory hierarchy adds small but fast memories (caches) near the CPU
  - Memories get bigger as distance from CPU increases
- CPU asks for data at address 1000
- Memory hierarchy anticipates more accesses to same address and stores a local copy
- Works well if CPU actually ends up using data from 1000 over and over and over …
- Otherwise wasted cache capacity
Reducing Artifactual Costs in Distributed Memory Architectures

- Data is transferred in chunks to amortize communication cost
  - Cell: DMA gets up to 16K
  - Usually get a contiguous chunk of memory

- Spatial locality
  - Computation should exhibit good spatial locality characteristics

- Temporal locality
  - Reorder computation to maximize use of data fetched
Single Thread Performance?
Single Thread Performance

- Tasks mapped to execution units (threads)
- Threads run on individual processors (cores)

Two keys to faster execution
- Load balance the work among the processors
- Make execution on each processor faster

Finish line: sequential time + longest parallel time
Understanding Performance

- Need some way of measuring performance
  - Coarse grained measurements
    - `% gcc sample.c
    - `% time a.out
      - 2.312u 0.062s 0:02.50 94.8%
    - `% gcc sample.c –O3
    - `% time a.out
      - 1.921u 0.093s 0:02.03 99.0%

- … but did we learn much about what’s going on?

```c
#define N (1 << 23)
#define T (10)
#include <string.h>
double a[N],b[N];

void cleara(double a[N]) {
    int i;
    for (i = 0; i < N; i++) {
        a[i] = 0;
    }
}

int main() {
    double s=0,s2=0; int i,j;
    for (j = 0; j < T; j++) {
        for (i = 0; i < N; i++) {
            b[i] = 0;
        }
        cleara(a);
        memset(a,0,sizeof(a));
        for (i = 0; i < N; i++) {
            s += a[i] * b[i];
            s2 += a[i] * a[i] + b[i] * b[i];
        }
    }
    printf("s %.2lf s2 %.2lf\n",s,s2);
}
```

record start time

record stop time
Hardware Performance Counters

- Special purpose registers built into the processor microarchitecture
  - Monitor hardware-related activity within the computer

- Useful for performance analysis and tuning
  - Provide low-level information that cannot be obtained with software profilers
Measurements Using Counters

- Increasingly possible to get accurate measurements using performance counters
  - Special registers in the hardware to measure events

- Insert code to start, read, and stop counter
  - Measure exactly what you want, anywhere you want
  - Can measure communication and computation duration
  - But requires manual changes
  - Monitoring nested scopes is an issue
  - Heisenberg effect: counters can perturb execution time
Dynamic Profiling

- **Event-based profiling**
  - Interrupt execution when an event counter reaches a threshold

- **Time-based profiling**
  - Interrupt execution every $t$ seconds

- **Works without modifying your code**
  - Does not require that you know where problem might be
  - Supports multiple languages and programming models
  - Quite efficient for appropriate sampling frequencies
Counter Examples

- Cycles (clock ticks)
- Pipeline stalls
- Cache hits
- Cache misses
- Number of instructions
- Number of loads
- Number of stores
- Number of floating point operations
- …
## Counters from Alpha EV5

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Reg#</th>
<th>Description</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLES</td>
<td>Total cycles</td>
<td>0, 2</td>
<td>ITB_MISS</td>
<td>2</td>
</tr>
<tr>
<td>ISSUES</td>
<td>Total issues</td>
<td>0</td>
<td>DCACHE_MISSES</td>
<td>2</td>
</tr>
<tr>
<td>NON_ISSUE_CYCLES</td>
<td>Nothing issued, pipeline frozen</td>
<td></td>
<td>DTB_MISS</td>
<td>2</td>
</tr>
<tr>
<td>SPLIT_ISSUE_CYCLES</td>
<td>Some but not all issuable instructions issued</td>
<td>1</td>
<td>LOADS_MERGED</td>
<td>2</td>
</tr>
<tr>
<td>PIPELINE_DRY</td>
<td>Nothing issued, pipeline dry</td>
<td>1</td>
<td>LDU_REPLAYS</td>
<td>2</td>
</tr>
<tr>
<td>REPLAY_TRAP</td>
<td>Replay traps (ldu, wb/maf, litmus test)</td>
<td>1</td>
<td>WB_MAF_FULL_REPLAYS</td>
<td>2</td>
</tr>
<tr>
<td>SINGLE_ISSUE_CYCLES</td>
<td>Single issue cycles</td>
<td>1</td>
<td>MEM_BARRIER</td>
<td>2</td>
</tr>
<tr>
<td>DUAL_ISSUE_CYCLES</td>
<td>Dual issue cycles</td>
<td>1</td>
<td>LOAD_LOCKED</td>
<td>2</td>
</tr>
<tr>
<td>TRIPLE_ISSUE_CYCLES</td>
<td>Triple issue cycles</td>
<td>1</td>
<td>SCACHE_ACCESS</td>
<td>1</td>
</tr>
<tr>
<td>QUAD_ISSUE_CYCLES</td>
<td>Quad issue cycles</td>
<td>1</td>
<td>SCACHE_READ</td>
<td>1</td>
</tr>
<tr>
<td>FLOW_CHANGE</td>
<td>Flow change (meaning depends on counter 2)</td>
<td>1</td>
<td>SCACHE_WRITE</td>
<td>1, 2</td>
</tr>
<tr>
<td>INTEGER_OPERATE</td>
<td>Integer operate instructions</td>
<td>1</td>
<td>SCACHE_VICTIM</td>
<td>1</td>
</tr>
<tr>
<td>FP_INSNS</td>
<td>FP operate instructions (not br, load, store)</td>
<td>1</td>
<td>SCACHE_MISS</td>
<td>2</td>
</tr>
<tr>
<td>LOAD_INSNS</td>
<td>Load instructions</td>
<td>1</td>
<td>SCACHE_READ_MISS</td>
<td>2</td>
</tr>
<tr>
<td>STORE_INSNS</td>
<td>Store instructions</td>
<td>1</td>
<td>SCACHE_WRITE_MISS</td>
<td>2</td>
</tr>
<tr>
<td>ICACHE_ACCESS</td>
<td>Instruction cache access</td>
<td>1</td>
<td>SCACHE_SH_WRITE</td>
<td>2</td>
</tr>
<tr>
<td>DCACHE_ACCESS</td>
<td>Data cache access</td>
<td>all</td>
<td>BCACHE_HIT</td>
<td>1</td>
</tr>
<tr>
<td>LONGSTALLS</td>
<td>Stalls longer than 15 cycles</td>
<td>2</td>
<td>BCACHE_VICTIM</td>
<td>1</td>
</tr>
<tr>
<td>PC_MISPR</td>
<td>PC mispredicts</td>
<td>2</td>
<td>BCACHE_MISS</td>
<td>2</td>
</tr>
<tr>
<td>BRANCH_MISPREDICTS</td>
<td>Branch mispredicts</td>
<td>2</td>
<td>SYS_REQ</td>
<td>1</td>
</tr>
<tr>
<td>ICACHE_MISSES</td>
<td>Instruction cache misses</td>
<td>2</td>
<td>SYS_INV</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SYS_READ_REQ</td>
<td>2</td>
</tr>
</tbody>
</table>
Sample counters from IBM Power 5++  
(759 counters total)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Reg#</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLES</td>
<td>Processor Cycles</td>
<td>1</td>
</tr>
<tr>
<td>CYCLES_RND_SMPL</td>
<td>Processor Cycles with random sampling</td>
<td>2</td>
</tr>
<tr>
<td>PM_RUN_CYC_GRP1</td>
<td>Run cycles</td>
<td>0</td>
</tr>
<tr>
<td>PM_INST_CMP_GRP1</td>
<td>Instructions completed</td>
<td>1</td>
</tr>
<tr>
<td>PM_INST_DISP_GRP1</td>
<td>Instructions dispatched</td>
<td>2</td>
</tr>
<tr>
<td>PM_CYC_GRP1</td>
<td>Processor cycles</td>
<td>3</td>
</tr>
<tr>
<td>PM_1PLUS_PPC_CMP_GRP2</td>
<td>One or more PPC instruction completed</td>
<td>0</td>
</tr>
<tr>
<td>PM_GCT_EMPTY_CYC_GRP2</td>
<td>Cycles GCT empty</td>
<td>1</td>
</tr>
<tr>
<td>PM_GRP_CMP_GRP2</td>
<td>Group completed</td>
<td>2</td>
</tr>
<tr>
<td>PM_CYC_GRP2</td>
<td>Processor cycles</td>
<td>3</td>
</tr>
<tr>
<td>PM_GRP_DISP_VALID_GRP3</td>
<td>Group dispatch valid</td>
<td>0</td>
</tr>
<tr>
<td>PM_GRP_DISP_REJ_GRP3</td>
<td>Group dispatch rejected</td>
<td>1</td>
</tr>
<tr>
<td>PM_GRP_DISP_BLK_SB_CYC_GRP3</td>
<td>Cycles group dispatch blocked by scoreboard</td>
<td>2</td>
</tr>
<tr>
<td>PM_INST_DISP_GRP3</td>
<td>Instructions dispatched</td>
<td>3</td>
</tr>
<tr>
<td>PM_0INST_CLB_CYC_GRP4</td>
<td>Cycles no instructions in CLB</td>
<td>0</td>
</tr>
<tr>
<td>PM_2INST_CLB_CYC_GRP4</td>
<td>Cycles 2 instructions in CLB</td>
<td>1</td>
</tr>
<tr>
<td>PM_CLB_EMPTY_CYC_GRP4</td>
<td>Cycles CLB empty</td>
<td>2</td>
</tr>
<tr>
<td>PM_MRK_DATA_FROM_L35_MOD_CYC_GRP4</td>
<td>Marked load latency from L3.5 modified</td>
<td>3</td>
</tr>
<tr>
<td>PM_5INST_CLB_CYC_GRP5</td>
<td>Cycles 5 instructions in CLB</td>
<td>0</td>
</tr>
<tr>
<td>PM_6INST_CLB_CYC_GRP5</td>
<td>Cycles 6 instructions in CLB</td>
<td>1</td>
</tr>
<tr>
<td>PM_MRK_LSU_SRQ_INST_VALID_GRP5</td>
<td>Marked instruction valid in SRQ</td>
<td>2</td>
</tr>
</tbody>
</table>

```  | PM_MRK_LSU_FLUSH_SRQ_GRP16 | Marked SRQ lhs flushes |
| PM_MRK_LSU0_REJECT_RELOAD_CDF_GRP17 | LSU0 reject due to reload CDF or tag update collision |
| PM_MRK_LSU1_REJECT_RELOAD_CDF_GRP17 | LSU1 reject due to reload CDF or tag update collision |
| PM_MRK_LSU0_REJECT_ERAT_MISS_GRP18 | LSU0 reject due to ERAT miss |
| PM_MRK_LSU1_REJECT_ERAT_MISS_GRP18 | LSU1 reject due to ERAT miss |
| PM_MRK_LWSYNC_HELD_GRP18 | LWSYNC held at dispatch |
| PM_MRK_LSU_FLUSH_SRQ_GRP16 | LSU0 reject due to reload CDF or tag update collision |
| PM_MRK_LSU0_REJECT_LMQ_FULL_GRP19 | LSU0 reject due to LMQ full or missed data coming |
| PM_MRK_LSU1_REJECT_LMQ_FULL_GRP19 | LSU1 reject due to LMQ full or missed data coming |
| PM_MRK_IOPS_CMP_GRP17 | Internal operations completed |
| PM_MRK_LSU_FLUSH_GRP20 | LSU SRQ lhs rejects |
| PM_MRK_LS0_FLUSH_GRP20 | LSU reject due to reload CDF or tag update collision |
| PM_MRK_FLUSH_GRP20 | Flush initiated by LSU |
| PM_MRK_FLUSH_GRP20 | Flushes |
| PM_MRK_FLUSH_GRP20 | Internal operations completed |
| PM_MRK_FLUSH_GRP20 | SRQ unaligned store flushes |
| PM_MRK_FLUSH_GRP20 | Flush caused by thread GCT imbalance |
| PM_MRK_DC_INV_L2_GRP21 | L1 D cache entries invalidated from L2 |
| PM_MRK_TLB_MISS_GRP22 | Instruction TLB misses |

(…)

Bilkent University
Useful Derived Measurements

- Processor utilization
  - Cycles / Wall Clock Time
- Instructions per cycle
  - Instructions / Cycles
- Instructions per memory operation
  - Instructions / Loads + Stores
- Average number of instructions per load miss
  - Instructions / L1 Load Misses
- Many others
  - Cache miss rate
  - Branch misprediction rate
  - …
Common Profiling Workflow

application source → compiler → binary object code

binary analysis

run (profiles execution) → performance profile

interpret profile → source correlation
Popular Runtime Profiling Tools

● GNU gprof
  ■ Widely available with UNIX/Linux distributions
    ```
    gcc -O2 -pg foo.c -o foo
    ./foo
    gprof foo
    ```

● HPC Toolkit
  ■ http://www.hipersoft.rice.edu/hpctoolkit/

● PAPI
  ■ http://icl.cs.utk.edu/papi/

● VTune

● Many others
**GNU gprof**

- **MPEG-2 decoder (reference implementation)**

  ```
  %./mpeg2decode -b mei16v2.m2v -f -r
  
  -r uses double precision inverse DCT
  ```

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>90.48</td>
<td>0.19</td>
<td>0.19</td>
<td>7920</td>
</tr>
<tr>
<td>4.76</td>
<td>0.20</td>
<td>0.01</td>
<td>2148</td>
</tr>
</tbody>
</table>

  ```
  %./mpeg2decode -b mei16v2.m2v -f
  
  uses fast integer based inverse DCT instead
  ```

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
</tr>
<tr>
<td>66.67</td>
<td>0.02</td>
<td>0.02</td>
<td>8238</td>
</tr>
<tr>
<td>33.33</td>
<td>0.03</td>
<td>0.01</td>
<td>63360</td>
</tr>
</tbody>
</table>
Baseline VTune Process View
Drill Down
Optimization Result

- Perform adds in parallel using wadd WMMX instr.

35% performance gain
Performance in Uniprocessors \( \text{time} = \text{compute} + \text{wait} \)

- Instruction level parallelism
  - Multiple functional units, deeply pipelined, speculation, ...

- Data level parallelism
  - SIMD: short vector instructions (multimedia extensions)
    - Hardware is simpler, no heavily ported register files
    - Instructions are more compact
    - Reduces instruction fetch bandwidth

- Complex memory hierarchies
  - Multiple level caches, may outstanding misses, prefetching, ...
**SIMD**

- Single Instruction, Multiple Data
- SIMD registers hold short vectors
- Instruction operates on all elements in SIMD register at once

Scalar code:
```c
for (int i = 0; i < n; i+=1) {
    c[i] = a[i] + b[i]
}
```

Vector code:
```c
for (int i = 0; i < n; i += 4) {
    c[i:i+3] = a[i:i+3] + b[i:i+3]
}
```
for (int i = 0; i < n; i+=1) {
    c[i] = a[i] + b[i]
}

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD</td>
<td>LOAD</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>STORE</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

Estimated cycles for loop:
- Scalar loop
  - n iterations * 3 cycles/iteration
SIMD Example

for (int i = 0; i < n; i+=4) {
    c[i:i+3] = a[i:i+3] + b[i:i+3]
}

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VLOAD</td>
<td>VLOAD</td>
</tr>
<tr>
<td>2</td>
<td>VADD</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VSTORE</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Estimated cycles for loop:
- **Scalar loop**
  n iterations * 3 cycles/iteration
- **SIMD loop**
  n/4 iterations * 3 cycles/iteration
- **Speedup:** ?
for (int i = 0; i < n; i+=4) {
    c[i:i+3] = a[i:i+3] + b[i:i+3]
}

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>VLOAD</td>
<td>VLOAD</td>
</tr>
<tr>
<td>2</td>
<td>VADD</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>VSTORE</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Estimated cycles for loop:
- **Scalar loop**
  - n iterations * 3 cycles/iteration
- **SIMD loop**
  - n/4 iterations * 3 cycles/iteration
- **Speedup**: 4x
SIMD in Major ISAs

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Architecture</th>
<th>SIMD Width</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>AltiVec</td>
<td>PowerPC</td>
<td>128</td>
<td>yes</td>
</tr>
<tr>
<td>MMX/SSE</td>
<td>Intel</td>
<td>64/128</td>
<td>yes</td>
</tr>
<tr>
<td>3DNow!</td>
<td>AMD</td>
<td>64</td>
<td>yes</td>
</tr>
<tr>
<td>VIS</td>
<td>Sun</td>
<td>64</td>
<td>no</td>
</tr>
<tr>
<td>MAX2</td>
<td>HP</td>
<td>64</td>
<td>no</td>
</tr>
<tr>
<td>MVI</td>
<td>Alpha</td>
<td>64</td>
<td>no</td>
</tr>
<tr>
<td>MDMX</td>
<td>MIPS V</td>
<td>64</td>
<td>yes</td>
</tr>
</tbody>
</table>

● And of course Cell
  ■ SPU has 128 128-bit registers
  ■ All instructions are SIMD instructions
  ■ Registers are treated as short vectors of 8/16/32-bit integers or single/double-precision floats
Performance in Uniprocessors \( \text{time} = \text{compute} + \text{wait} \)

- **Instruction level parallelism**
  - Multiple functional units, deeply pipelined, speculation, ...

- **Data level parallelism**
  - SIMD: short vector instructions (multimedia extensions)
    - Hardware is simpler, no heavily ported register files
    - Instructions are more compact
    - Reduces instruction fetch bandwidth

- **Complex memory hierarchies**
  - Multiple level caches, may outstanding misses, prefetching, ...
  - Exploiting locality is essential
Instruction Locality

Baseline

for i = 1 to N
    A();
    B();
    C();
end

Baseline miss rate = 1

cache miss

cache hit
### Instruction Locality

#### Cache Miss

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Full Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i = 1 to N</td>
<td>for i = 1 to N</td>
</tr>
<tr>
<td>A();</td>
<td>A();</td>
</tr>
<tr>
<td>B();</td>
<td>B();</td>
</tr>
<tr>
<td>C();</td>
<td>C();</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td><strong>cache miss</strong></td>
<td><strong>cache hit</strong></td>
</tr>
</tbody>
</table>

**Working Set Size**

- **Baseline**: miss rate = 1
- **Full Scaling**: miss rate = 1 / N

**Cache Size**

- A
- B
- C

---

**Diagram:**

- For loop from 1 to N
- Operations A, B, C
- Cache miss and hit indicators

---

**Working Set Size:**

- Cache miss rate calculation
- Baseline vs Full Scaling comparison
Example Memory (Cache) Optimization

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Full Scaling</th>
</tr>
</thead>
</table>
| for i = 1 to N  
A();  
B();  
C();  
end | for i = 1 to N  
A();  
for i = 1 to N  
B();  
for i = 1 to N  
C(); |

**Working Set Size**

<table>
<thead>
<tr>
<th>cache size</th>
<th>inst</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A + B + C</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A + B + C</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>inst</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>
# Example Memory (Cache) Optimization

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Full Scaling</th>
<th>Full Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i = 1 to N</td>
<td>for i = 1 to N</td>
<td>for i = 1 to N</td>
</tr>
<tr>
<td>A();</td>
<td>A();</td>
<td>A();</td>
</tr>
<tr>
<td>B();</td>
<td>B();</td>
<td>B();</td>
</tr>
<tr>
<td>C();</td>
<td>for i = 1 to N</td>
<td>for i = 1 to N</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

![Diagram](image_url)
**Example Memory (Cache) Optimization**

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Full Scaling</th>
<th>Cache Aware</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i = 1 to N</td>
<td><code>A();</code></td>
<td>for i = 1 to N</td>
<td>for j = 1 to 64</td>
</tr>
<tr>
<td></td>
<td><code>A();</code></td>
<td>for i = 1 to N</td>
<td><code>A();</code></td>
</tr>
<tr>
<td></td>
<td><code>B();</code></td>
<td>for i = 1 to N</td>
<td><code>B();</code></td>
</tr>
<tr>
<td></td>
<td><code>C();</code></td>
<td>for i = 1 to N</td>
<td><code>C();</code></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>

**Working Set Size**

- **Cache size**
- **Inst**
- **Data**

<table>
<thead>
<tr>
<th></th>
<th>Inst</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>
Results from Cache Optimizations

Reuse and Locality

- Consider how data is accessed
  - **Data reuse:**
    - Same data used multiple times
    - Intrinsic in computation
  - **Data locality:**
    - Data is reused and is present in “fast memory”
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations
Temporal Reuse in Sequential Code

• Same data used in distinct iterations I and I'

```c
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
```

• $A[j]$ has self-temporal reuse in loop i
Spatial Reuse

- Same data transfer (usually cache line) used in distinct iterations I and I'

for (i=1; i<N; i++)
    for (j=1; j<N; j++)

- A[j] has self-spatial reuse in loop j
- **Multi-dimensional array note**: C arrays are stored in row-major order
Group Reuse

- Same data used by distinct references

```c
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
```

- `A[j], A[j+1] and A[j-1]` have group reuse (spatial and temporal) in loop `j`
Cache Locality

• Suppose array $A$ has column-major layout


• Loop nest has poor spatial cache locality.

```plaintext
for i = 1, 100
  for j = 1, 200
  end_for
end_for
```
Loop Interchange

• Suppose array A has column-major layout

\[
\begin{array}{cccccc}
\end{array}
\]

for \( i = 1, 100 \)

    for \( j = 1, 200 \)
    end_for
end_for

• New loop nest has \textbf{better} spatial cache locality.
Interchange Loops?

\[
\begin{align*}
\text{for } i &= 2, 100 \\
\text{for } j &= 1, 200 \\
A[i, j] &= A[i-1, j+1] + 3 \\
\text{end}_\text{for} \\
\text{end}_\text{for}
\end{align*}
\]

- e.g. dependence from (3,3) to (4,2)
Dependence Vectors

- Distance vector \((1,-1) = (4,2)-(3,3)\)
- Direction vector \((+, -)\) from the signs of distance vector
- Loop interchange is not legal if there exists dependence \((+, -)\)
Safety of Permutation

- **Intuition:** Cannot permute two loops $i$ and $j$ in a loop nest if doing so changes the relative order of a read and write or two writes to the same memory location.

```
for (i = 0; i < 3; i++)
    for (j = 0; j < 6; j++)
```

- Ok to permute?

```
for (i = 0; i < 3; i++)
    for (j = 0; j < 6; j++)
```
Loop Interchange Example

- Consider the (<,>) case

**Before**
do i = 1,n
do j = 1,n
  \[ C(i,j) = C(i+1,j-1) \]
enddo
endo
do j = 1,n
do i = 1,n
  \[ C(i,j) = C(i+1,j-1) \]
enddo
endo

\[
\begin{align*}
(1,1) & \quad C(1,1) = C(2,0) \\
(1,2) & \quad C(1,2) = C(2,1) \\
\cdots & \quad \cdots \\
(2,1) & \quad C(2,1) = C(3,0)
\end{align*}
\]

**After**
do j = 1,n
do i = 1,n
  \[ C(i,j) = C(i+1,j-1) \]
enddo
endo
ndo j = 1,n
do i = 1,n
  \[ C(i,j) = C(i+1,j-1) \]
enddo
endo

\[
\begin{align*}
(1,1) & \quad C(1,1) = C(2,0) \\
(2,1) & \quad C(2,1) = C(3,0) \\
\cdots & \quad \cdots \\
(1,2) & \quad C(1,2) = C(2,1)
\end{align*}
\]
Loop Interchange - Legality

- $(=,=)$
  - The dependence is loop independent, so it is unaffected by interchange.

- $(=,<)$
  - The dependence is carried by the $j$ loop.
  - After interchange the dependence will be $(<,=)$, so the dependence will still be carried by the $j$ loop, so the dependence relations do not change.

- $(<,=)$
  - The dependence is carried by the $i$ loop.
  - After interchange the dependence will be $(=,<)$, so the dependence will still be carried by the $i$ loop, so the dependence relations do not change.

- $(<,<)$
  - The dependence distance is positive in both dimensions.
  - After interchange it will still be positive in both dimensions, so the dependence relations do not change.

- $(<,>)$
  - The dependence is carried by the outer loop.
  - After interchange the dependence will be $(>,<)$, which changes the dependences and results in an illegal direction vector, so interchange is illegal.
Loop Fusion

- Better reuse between $A[i]$ and $A[i]$
Loop Distribution

- 2\textsuperscript{nd} loop is parallel

```plaintext
for i = 1, 1000
end_for

for i = 1, 1000
    C[i] = B[i] + 5
end_for
```
Register Blocking

for \( j = 1, 2^m \)
   for \( i = 1, 2^n \)
   end_for
end_for

for \( j = 1, 2^m, 2 \)
   for \( i = 1, 2^n, 2 \)
   end_for
end_for

- Better reuse between \( A[i,j] \) and \( A[i,j] \)
Virtual Register Allocation

for \( j = 1, 2^*M, 2 \)
for \( i = 1, 2^*N, 2 \)
    \( r1 = A[i-1,j] \)
    \( r2 = r1 + A[i-1,j-1] \)
    \( A[i, j] = r2 \)
    \( r3 = A[i-1,j+1] + r1 \)
    \( A[i, j+1] = r3 \)
    \( A[i+1, j] = r2 + A[i, j-1] \)
    \( A[i+1, j+1] = r3 + r2 \)
end_for
end_for

- Memory operations reduced to register load/store
- 8MN loads to 4MN loads
Scalar Replacement

- Eliminate loads and stores for array references

```c
for i = 2, N+1
   t1 = A[i-1] + 1
   A[i] = t1
end_for
```
Loop Unrolling for ILP

- Large scheduling regions. Fewer dynamic branches
- Increased code size

```
for i = 1, 10
    a[i] = b[i];
    *p = ...
end_for
```

```
for I = 1, 10, 2
    a[i] = b[i];
    *p = ...
    a[i+1] = b[i+1];
    *p = ...
end_for
```
Tiling Example

for (j=1; j<M; j++)
    for (i=1; i<N; i++)
        D[i] = D[i] + B[j][i];

Strip mine

for (j=1; j<M; j++)
    for (ii=1; ii<N; ii+=s)
        for (i=ii; i<min(ii+s-1,N); i++)
            D[i] = D[i] + B[j][i];

Permute

for (ii=1; ii<N; ii+=s)
    for (j=1; j<M; j++)
        for (i=ii; i<min(ii+s-1,N); i++)
            D[i] = D[i] + B[j][i];
Tiling (Blocking): Another Loop Reordering Transformation

- Tiling reorders loop iterations to bring iterations that reuse data closer in time.
Naive Matrix Multiply

\[
\{\text{implements } C = C + A \times B\}
\]

for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      \[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \]

Algorithm has \(2n^3 = O(n^3)\) Flops and operates on \(3n^2\) words of memory

Reuse quotient (\(q = \text{flops/word}\)) in the algorithm is \(\text{potentially as large as}\)

\[
\frac{2n^3}{3n^2} = O(n)
\]
Naive Matrix Multiply

\{\text{implements } C = C + A^*B\}\}

for \(i = 1\) to \(n\)
\{\text{read row } i \text{ of } A \text{ into fast memory}\}

for \(j = 1\) to \(n\)
\{\text{read } C(i,j) \text{ into fast memory}\}
\{\text{read column } j \text{ of } B \text{ into fast memory}\}

for \(k = 1\) to \(n\)
\[C(i,j) = C(i,j) + A(i,k) \times B(k,j)\]
\{\text{write } C(i,j) \text{ back to slow memory}\}
Consider $A, B, C$ to be $N$-by-$N$ matrices of $b$-by-$b$ subblocks where $b = n / N$ is called the block size.

for $i = 1$ to $N$
  for $j = 1$ to $N$
    {read block $C(i,j)$ into fast memory}
    for $k = 1$ to $N$
      {read block $A(i,k)$ into fast memory}
      {read block $B(k,j)$ into fast memory}
      \[ C(i,j) = C(i,j) + A(i,k) \times B(k,j) \] \{do a matrix multiply on blocks\}
    {write block $C(i,j)$ back to slow memory}
Programming for Performance

● Tune the parallelism first

● Then tune performance on individual processors
  ■ Modern processors are complex
  ■ Need instruction level parallelism for performance
  ■ Understanding performance requires a lot of probing

● Optimize for the memory hierarchy
  ■ Memory is much slower than processors
  ■ Multi-layer memory hierarchies try to hide the speed gap
  ■ Data locality is essential for performance
Programming for Performance

● May have to change everything!
  ■ Algorithms, data structures, program structure

● Focus on the biggest performance impediments
  ■ Too many issues to study everything
  ■ Remember the law of diminishing returns
Additional Tiling Material
Legality of Tiling

- Tiling is safe only if it does not change the order in which memory locations are read/written
  - We’ll talk about correctness after memory hierarchies
- Tiling can conceptually be used to perform the decomposition into threads and blocks
  - We’ll show this later, too
A Few Words On Tiling

• Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in constant cache exceeds cache capacity or data in registers exceeds register capacity or (as in example) data in shared memory for block still exceeds shared memory capacity
Matrix Multiply in CUDA

- Imagine you want to compute extremely large matrices.
  - That don’t fit in global memory
- This is where an additional level of tiling could be used, between grids
CUDA Version of Example (Tiling for Computation Partitioning)

for (ii=1; ii<N; ii+=s)
  for (i=ii; i<min(ii+s-1,N); i++)
    for (j=1; j<N; j++)
      \[ D[i] = D[i] + B[j][i]; \]

...  
\[ <<<\text{ComputeI}(N/s,s)>>>(d\_D, d\_B, N); \]
...

\[
\text{__global__ ComputeI (float *d\_D, float *d\_B, int N) \{ }
\]
  \[
i \text{int ii = blockIdx.x;}
\]
  \[
i \text{int i = ii*s + threadIdx.x;}
\]
  \[
i \text{for (j=0; j<N; j++)}
\]
  \[
i \text{d\_D[i] = d\_D[i] + d\_B[j*N+i];}
\]
\[
\text{\}}
\]
Tiled Matrix Multiply Using Thread Blocks

- One block computes one square sub-matrix $P_{\text{sub}}$ of size BLOCK_SIZE.
- One thread computes one element of $P_{\text{sub}}$.
- Assume that the dimensions of $M$ and $N$ are multiples of BLOCK_SIZE and square shape.

![Diagram showing the tiled matrix multiply using thread blocks.](image)
CUDA Code - Kernel Execution Configuration

// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.width / dimBlock.x,
              M.height / dimBlock.y);

For very large N and M dimensions, one will need to add another level of blocking and execute the second-level blocks sequentially.
CUDA Code - Kernel Overview

// Block index
int bx = blockIdx.x;
int by = blockIdx.y;

// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;

// Pvalue stores the element of the block sub-matrix
// that is computed by the thread
float Pvalue = 0;

// Loop over all the sub-matrices of M and N
// required to compute the block sub-matrix
for (int m = 0; m < M.width/BLOCK_SIZE; ++m) {
    code from the next few slides
}
// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);

// Get a pointer to the current sub-matrix Nsub of N
Matrix Nsub = GetSubMatrix(N, bx, m);

__shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

// each thread loads one element of the sub-matrix
Ms[ty][tx] = GetMatrixElement(Msub, tx, ty);

// each thread loads one element of the sub-matrix
Ns[ty][tx] = GetMatrixElement(Nsub, tx, ty);
// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();

// each thread computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of M and N in the next iteration
__syncthreads();
CUDA Code - Save Result

// Get a pointer to the block sub-matrix of P
Matrix Psub = GetSubMatrix(P, bx, by);

// Write the block sub-matrix to device memory;
// each thread writes one element
SetMatrixElement(Psub, tx, ty, Pvalue);

This code should run at about 150 Gflops on a GTX or Tesla.

State-of-the-art mapping (in CUBLAS 3.2 on C2050) yields just above 600 Gflops. Higher on GTX480.