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

synchronization point

CPU is idle

Memory is idle
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
- 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
Need some way of measuring performance

- Coarse grained measurements
  
  \[
  \text{% gcc sample.c} \\
  \text{% time a.out} \\
  2.312u 0.062s 0:02.50 94.8\% \\
  \text{% gcc sample.c -O3} \\
  \text{% time a.out} \\
  1.921u 0.093s 0:02.03 99.0\% \\
  \]

- ... 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));
  record start time

  for (i = 0; i < N; i++) {
    s += a[i] * b[i];
    s2 += a[i] * a[i] + b[i] * b[i];
  }

  record stop time
  printf("s %f s2 %f\n", s, s2);
}
```
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 can not 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
- ...

Bilkent University
## Counters from Alpha EV5

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Reg#</th>
<th>ITB_MISS</th>
<th>DCACHE_MISSES</th>
<th>LOADS_MERGED</th>
<th>LDU_REPLAYS</th>
<th>WB_MAF_FULL_REPLAYS</th>
<th>MEM_BARRIER</th>
<th>Instruction TLB miss</th>
<th>Data cache misses</th>
<th>Loads merged in MAF</th>
<th>LDU replay traps</th>
<th>WB/MAF full replay traps</th>
<th>Memory barrier instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLES</td>
<td>Total cycles</td>
<td>0, 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ISSUES</td>
<td>Total issues</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NON_ISSUE_CYCLES</td>
<td>Nothing issued, pipeline frozen</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPLIT_ISSUE_CYCLES</td>
<td>Some but not all issuable instructions issued</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIPELINE_DRY</td>
<td>Nothing issued, pipeline dry</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPLAY_TRAP</td>
<td>Replay traps (ldu, wb/maf, litmus test)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE_ISSUE_CYCLES</td>
<td>Single issue cycles</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUAL_ISSUE_CYCLES</td>
<td>Dual issue cycles</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIPLE_ISSUE_CYCLES</td>
<td>Triple issue cycles</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUAD_ISSUE_CYCLES</td>
<td>Quad issue cycles</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOW_CHANGE</td>
<td>Flow change (meaning depends on counter 2)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER_OPERATE</td>
<td>Integer operate instructions</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP_INSNS</td>
<td>FP operate instructions (not br, load, store)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD_INSNS</td>
<td>Load instructions</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE_INSNS</td>
<td>Store instructions</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICACHE_ACCESS</td>
<td>Instruction cache access</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCACHE_ACCESS</td>
<td>Data cache access</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LONG_STALLS</td>
<td>Stalls longer than 15 cycles</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC_MISPR</td>
<td>PC mispredicts</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BRANCH_MISPREDICTS</td>
<td>Branch mispredicts</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICACHE_MISSES</td>
<td>Instruction cache misses</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key Definitions:**
- **ITB_MISS:** Instruction TLB misses
- **DCACHE_MISSES:** Data cache misses
- **LOADS_MERGED:** Loads merged in MAF
- **LDU_REPLAYS:** LDU replay traps
- **WB_MAF_FULL_REPLAYS:** WB/MAF full replay traps
- **MEM_BARRIER:** Memory barrier instructions
- **LOAD_LOCKED:** LDx/L instructions
- **SCACHE_ACCESS:** S-cache access
- **SCACHE_READ:** S-cache read
- **SCACHE_WRITE:** S-cache write
- **SCACHE_VICTIM:** S-cache victim
- **SCACHE_MISS:** S-cache miss
- **SCACHE_READ_MISS:** S-cache read miss
- **SCACHE_WRITE_MISS:** S-cache write miss
- **SCACHE_SH_WRITE:** S-cache shared writes
- **BCACHE_HIT:** B-cache hit
- **BCACHE_VICTIM:** B-cache victim
- **BCACHE_MISS:** B-cache miss
- **SYS_REQ:** System requests
- **SYS_INV:** System invalidates
- **SYS_READ_REQ:** System read requests
<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_CMPL_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_CMPL_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_CMPL_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_REJECT_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_MRK_LSU_SRQ_INST_VALID_GRP5</td>
<td>Marked instruction valid in SRQ</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
<th>Reg#</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_MRK_LSU_FLUSH_SRQ_GRP16</td>
<td>Marked SRQ lhs flushes</td>
</tr>
<tr>
<td>PM_LSU0_REJECT_RELOAD_CDF_GRP17</td>
<td>LSU0 reject due to reload CDF or tag update collision</td>
</tr>
<tr>
<td>PM_LSU1_REJECT_RELOAD_CDF_GRP17</td>
<td>LSU1 reject due to reload CDF or tag update collision</td>
</tr>
<tr>
<td>PM_IOSPS_CMPL_GRP17</td>
<td>Internal operations completed</td>
</tr>
<tr>
<td>PM_L1_WRITE_CYC_GRP17</td>
<td>Cycles writing to instruction L1</td>
</tr>
<tr>
<td>PM_LSU0_REJECT_ERAT_MISS_GRP18</td>
<td>LSU0 reject due to ERAT miss</td>
</tr>
<tr>
<td>PM_LSU1_REJECT_ERAT_MISS_GRP18</td>
<td>LSU1 reject due to ERAT miss</td>
</tr>
<tr>
<td>PM_LWSYNC_HELD_GRP18</td>
<td>LWSYNC held at dispatch</td>
</tr>
<tr>
<td>PM_TLBIE_HELD_GRP18</td>
<td>TLBIE held at dispatch</td>
</tr>
<tr>
<td>PM_LSU0_REJECT_LMQ_FULL_GRP19</td>
<td>LSU0 reject due to LMQ full or missed data coming</td>
</tr>
<tr>
<td>PM_LSU1_REJECT_LMQ_FULL_GRP19</td>
<td>LSU1 reject due to LMQ full or missed data coming</td>
</tr>
<tr>
<td>PM_IOSPS_CMPL_GRP19</td>
<td>Internal operations completed</td>
</tr>
<tr>
<td>PM_BR_ISSUED_GRP19</td>
<td>Branches issued</td>
</tr>
<tr>
<td>PM_LSU0_REJECT_SRQ_GRP20</td>
<td>LSU SRQ lhs rejects</td>
</tr>
<tr>
<td>PM_LSU1_REJECT_SRQ_GRP20</td>
<td>LSU reject due to SRQ lhs rejects</td>
</tr>
<tr>
<td>PM_IOSPS_CMPL_GRP21</td>
<td>Internal operations completed</td>
</tr>
<tr>
<td>PM_LSU_FLUSH_GRP20</td>
<td>Flush initiated by LSU</td>
</tr>
<tr>
<td>PM_FLUSH_GRP20</td>
<td>Flushes</td>
</tr>
<tr>
<td>PM_IOSPS_CMPL_GRP21</td>
<td>Internal operations completed</td>
</tr>
<tr>
<td>PM_LSU_FLUSH_UST_GRP21</td>
<td>SRQ unaligned store flushes</td>
</tr>
<tr>
<td>PM_LSU_FLUSH_IMBAL_GRP21</td>
<td>Flush caused by thread GCT imbalance</td>
</tr>
<tr>
<td>PM_DC_INV_L2_GRP21</td>
<td>L1 D cache entries invalidated from L2</td>
</tr>
<tr>
<td>PM_TLB_INV_L2_GRP22</td>
<td>Instruction TLB misses</td>
</tr>
</tbody>
</table>
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

source correlation

binary analysis

run (profiles execution)

interpret profile

performance profile
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/](http://www.hipersoft.rice.edu/hpctoolkit/)

- **PAPI**

- **VTune**

- **Many others**
**GNU gprof**

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

  ```
  %./mpeg2decode -b mei16v2.m2v -f -r
  -r uses double precision inverse DCT
  
  %   cumulative   self              self     total
  time   seconds   seconds    calls  ns/call  ns/call  name
  90.48      0.19     0.19     7920 23989.90 23989.90  Reference_IDCT
  4.76       0.20     0.01     2148  4655.49  4655.49  Decode_MPEG1_Intra_Block
  
  %   cumulative   self              self     total
  time   seconds   seconds    calls  ns/call  ns/call  name
  66.67      0.02     0.02     8238  2427.77  2427.77  form_component_prediction
  33.33      0.03     0.01    63360  157.83   157.83  idctcol
  ```

  `%./mpeg2decode -b mei16v2.m2v -f`

  uses fast integer based inverse DCT instead
Baseline VTune Process View
Drill Down
Optimization Result

- Perform adds in parallel using wadd WMMX instr.

35% performance gain
Performance in Uniprocessors

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

Vector code:
```
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**

```c
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
  \[ n \text{ iterations} \times 3 \text{ cycles/iteration} \]
- SIMD loop
  \[ \frac{n}{4} \text{ iterations} \times 3 \text{ cycles/iteration} \]
- 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 time = compute + 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

miss rate = 1

.cache miss

.cache hit
## Instruction Locality

### Baseline

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

### Full Scaling

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

- **Cache Miss Rate**
  - Baseline: 1
  - Full Scaling: $1 / N$

- **Cache Size**
  - Working Set Size

- **Cache Miss**
  - Baseline: X
  - Full Scaling: ✓ ✓ ✓ ✓ ✓ ...

- **Working Set Size**
  - Cache Size

---

Bilkent University
Example Memory (Cache) Optimization

<table>
<thead>
<tr>
<th></th>
<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>
<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>C();</td>
<td>C();</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</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();
```
## Example Memory (Cache) Optimization

### Working Set Size

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Inst</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A+B/C</td>
<td></td>
</tr>
</tbody>
</table>

### Baseline

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

### Full Scaling

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

### Table

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

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

- **Baseline**: Iterates through each task sequentially.
- **Full Scaling**: Scales down the problem size to fit in cache.
- **Cache Aware**: Further optimizes by considering cache size and working set size.
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 $I'$

```plaintext
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 \( I' \)

```c
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


• New loop nest has better spatial cache locality.

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

for j = 1, 200
    for i = 1, 100
    end_for
end_for
```
Interchange Loops?

\begin{verbatim}
for i = 2, 100
  for j = 1, 200
  end_for
end_for
\end{verbatim}

- 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.

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

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

- **Ok to permute?**
Loop Interchange Example

- Consider the (<,>) case

**Before**

do i = 1,n
    do j = 1,n
        C(i,j) = C(i+1,j-1)
    enddo
enddo

(1,1) \( C(1,1) = C(2,0) \)
(1,2) \( C(1,2) = C(2,1) \)
...  
(2,1) \( C(2,1) = C(3,0) \)

**After**

do j = 1,n
    do i = 1,n
        C(i,j) = C(i+1,j-1)
    enddo
enddo

(1,1) \( C(1,1) = C(2,0) \)
(2,1) \( C(2,1) = C(3,0) \)
...  
(1,2) \( C(1,2) = C(2,1) \)
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 $A[i]$

```plaintext
for i = 1, 1000
    A[i] = B[i] + 3
end_for

for j = 1, 1000
end_for
```
Loop Distribution

for i = 1, 1000
end_for

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

- 2nd loop is parallel
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

```
for i = 2, N+1
  = A[i-1] + 1
  A[i] =
end_for
```

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

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

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

```c
for I = 1, 10, 2
    a[i] = b[i];
    *p = ...
    a[i+1] = b[i+1];
    *p = ...
end_for
```
for (j=1; j<M; j++)
    for (i=1; i<N; i++)
        D[i] = D[i] + B[j][i];

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];

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

{implements $C = C + A*B$}
for $i = 1$ to $n$
  for $j = 1$ to $n$
    for $k = 1$ to $n$
      $C(i,j) = C(i,j) + A(i,k) \cdot B(k,j)$

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

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

$$2*n^3 / 3*n^2 = O(n)$$
Naive Matrix Multiply

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

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 \( n \)

\[
C(i,j) = C(i,j) + A(i,k) \times B(k,j)
\]

\[
\{ \text{write } C(i,j) \text{ back to slow memory} \}
\]
Naïve Matrix Multiply

- Number of slow memory references on unblocked matrix multiply
  \[ m = n^3 \] to read each column of B \( n \) times
  \[ + n^2 \] to read each row of A once
  \[ + 2n^2 \] to read and write each element of C once
  \[ = n^3 + 3n^2 \]

- So the re-use quotient, \( q = f / m = 2n^3 / (n^3 + 3n^2) \)
  \[ \approx 2 \] for large \( n \)

- This is no better than matrix-vector multiply
- And is far from the “best possible” which is \( 2/3 * n \) for large \( n \)
- And this doesn’t take into account cache lines
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) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}
Blocked (Tiled) Matrix Multiply

- Recall:
  - $m$ is amount memory traffic between slow and fast memory
  - matrix has $nxn$ elements, also $NxN$ blocks each of size $bxb$ ($N=n/b$)
  - $f$ is number of floating point operations, $2n^3$ for this problem
  - $q = f / m$ is our measure of algorithm efficiency in the memory system

\[
m = N*n^2 \quad \text{read each block of B \ N^3 times (N^3 * b^2 = N^3 * (n/N)^2 = N*n^2)}
\]
\[
+ N*n^2 \quad \text{read each block of A \ N^3 times}
\]
\[
+ 2n^2 \quad \text{read and write each block of C once}
\]
\[
= (2N + 2) * n^2
\]

- Re-use quotient is:
  \[
  q = f / m = 2n^3 / ((2N + 2) * n^2)
  \]
  \[
  \approx n / N = b \quad \text{for large n}
  \]

- We can improve performance by increasing the blocksize $b$
  - Can be much faster than matrix-vector multiply ($q=2$)
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];

...<<<<ComputeI(N/s,s)>>>(d_D, d_B, N);
...

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

- **One block** computes one square sub-matrix $P_{\text{sub}}$ of size $\text{BLOCK\_SIZE}$
- **One thread** computes one element of $P_{\text{sub}}$
- Assume that the dimensions of $M$ and $N$ are multiples of $\text{BLOCK\_SIZE}$ and square shape
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.
Using Analysis to Understand Machines

The blocked algorithm has computational intensity \( q \approx b \)

- The larger the block size, the more efficient our algorithm will be
- Limit: All three blocks from A, B, C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large
- Assume your fast memory has size \( M_{\text{fast}} \)

\[
3b^2 \leq M_{\text{fast}}, \text{ so } q \approx b \leq \sqrt{M_{\text{fast}}/3}
\]

- To build a machine to run matrix multiply at 1/2 peak arithmetic speed of the machine, we need a fast memory of size \( M_{\text{fast}} \geq 3b^2 \approx 3q^2 = 3(t_m/t_f)^2 \)
- This size is reasonable for L1 cache, but not for register sets
- Note: analysis assumes it is possible to schedule the instructions perfectly

<table>
<thead>
<tr>
<th></th>
<th>( t_m/t_f )</th>
<th>required KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultra 2i</td>
<td>24.8</td>
<td>14.8</td>
</tr>
<tr>
<td>Ultra 3</td>
<td>14</td>
<td>4.7</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>6.25</td>
<td>0.9</td>
</tr>
<tr>
<td>Pentium3M</td>
<td>10</td>
<td>2.4</td>
</tr>
<tr>
<td>Power3</td>
<td>8.75</td>
<td>1.8</td>
</tr>
<tr>
<td>Power4</td>
<td>15</td>
<td>5.4</td>
</tr>
<tr>
<td>Itanium1</td>
<td>36</td>
<td>31.1</td>
</tr>
<tr>
<td>Itanium2</td>
<td>5.5</td>
<td>0.7</td>
</tr>
</tbody>
</table>
Limits to Optimizing Matrix Multiply

- The blocked algorithm changes the order in which values are accumulated into each $C[i,j]$ by applying associativity
  - Get slightly different answers from naïve code, because of roundoff - OK
- The previous analysis showed that the blocked algorithm has computational intensity:
  \[ q \approx b \leq \sqrt{\frac{M_{\text{fast}}}{3}} \]

- Aside (for those who have taken CS170 or equivalent)
  - There is a lower bound result that says we cannot do any better than this (using only associativity)

- Theorem (Hong & Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to $q = O(\sqrt{M_{\text{fast}}})$

- What if more levels of memory hierarchy?
Tiling for Multiple Levels

- Multiple levels: pages, caches, registers in machines
- Each level could have its own 3-nested loops:
  
  ```
  for i, for j, for k {matmul blocks that fit in L2 cache}
  for ii, for jj, for kk {matmul blocks that fit in L1}
  for iii, for jjj, for kkk {matmul blocks that fit in registers}
  ```

- But in practice don’t use so many levels and fully unroll code for register “tiles”
  - E.g., if these are 2x2 matrix multiplies, can write out all 4 statements without loops:
    ```
    c[0,0] = c[0,0] + a[0,0] * b[0,0] + a[0,1] * b[1,0]
    c[1,0] = c[0,1] = c[1,1] =
    ```

  Many possible code variations; see Mark’s notes on code generators in HW page.
Matrix multiply, optimized several ways

Speed of n-by-n matrix multiply on Sun Ultra-1/170, peak = 330 MFlops

9/10/2007 CS194 Lecture
Search Over Block Sizes

Strategies for choose block sizes:

● Find out hardware parameters:
  ■ Read vendor manual for register #, cache sizes (not part of ISA), page sizes (OS config)
  ■ Measure yourself using memory benchmark from last time
  ■ But in practice these don’t always work well; memory systems are complex as we saw

● Manually try many variations 😞

● Write “autotuner” to search over “design space” of possible implementations
  ■ Atlas – incorporated into Matlab
  ■ PhiPAC – original dense linear algebra tuning project from Berkeley; came from homework assignment like HW2
What the Search Space Looks Like

A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.

(Platform: Sun Ultra-III, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

9/10/2007 CS194 Lecture
ATLAS (DGEMM n = 500)

Source: Jack Dongarra

- ATLAS is faster than all other portable BLAS implementations and it is comparable with machine-specific libraries provided by the vendor.
Recursion: Cache Oblivious Algorithms

- The tiled algorithm requires finding a good block size
- Cache Oblivious Algorithms offer an alternative
  - Treat n x n matrix multiply set of smaller problems
  - Eventually, these will fit in cache

- The idea of cache-oblivious algorithms is use for other problems than matrix multiply. The general idea is:
  - Think of recursive formulation
  - If the subproblems use smaller data sets and some reuse within that data set, then a recursive order may improve performance
Cache-Oblivious Algorithms

\[
\begin{array}{cc}
B_{00} & B_{01} \\
B_{10} & B_{11} \\
\end{array}
\]

\[
\begin{array}{cc}
A_{00} & A_{01} \\
A_{10} & A_{11} \\
\end{array}
\]

\[
\begin{array}{cc}
C_{00} & C_{01} \\
C_{10} & C_{11} \\
\end{array}
\]

\[
\begin{array}{cc}
A_0 & C_0 \\
A_1 & C_1 \\
\end{array}
\]

\[
\begin{align*}
C_{00} &= A_{00} * B_{00} + A_{01} * B_{10} \\
C_{01} &= A_{01} * B_{11} + A_{00} * B_{01} \\
C_{11} &= A_{11} * B_{01} + A_{10} * B_{01} \\
C_{10} &= A_{10} * B_{00} + A_{11} * B_{10} \\
\end{align*}
\]

- Divide all dimensions (AD)
- 8-way recursive tree down to 1x1 blocks
  - Gray-code order promotes reuse
- Bilardi, et al.

\[
\begin{align*}
C_0 &= A_0 * B \\
C_1 &= A_1 * B \\
\end{align*}
\]

- Divide largest dimension (LD)
- Two-way recursive tree down to 1x1 blocks

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Block sizes
- Generated dynamically at each level in the recursive call tree

Our experience
- Performance is similar
- Use AD for the rest of the talk

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Cache-Conscious Algorithms

- Usually Iterative
  - Nested loops

- Implementation of blocking
  - Cache blocking achieved by Loop Tiling
  - Register blocking also requires Loop Unrolling
Structure of Tiled (Cache Concious) Code

Cache Blocking

Register Blocking

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Data Structures

- Fit control structure better
- Improve
  - Spatial locality
  - Streaming, prefetching

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Data Structures: Discussion

- Morton-Z
  - Matches recursive control structure better than RBR
  - Suggests better performance for CO
  - More complicated to implement
  - In our experience payoff is small or even negative
    - Use RBR for the rest of the talk

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
UltraSPARC IIIi

- **Peak performance**
  - 2 GFlops

- **Memory hierarchy**
  - Registers: 32
  - L1 data cache: 64KB, 4-way
  - L2 data cache: 1MB, 4-way

- **Compilers**
  - FORTRAN: SUN F95 7.1
  - C: SUN C 5.5

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Control Structures

- Iterative: triply nested loop
- Recursive: down to $1 \times 1 \times 1$

[Diagram of control structures with iterative and recursive branches]

滑源：Roeder, Yotov, Pingali at Cornell/UTexas

9/10/
Control Structures

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive
- Micro-Kernel

None / Compiler

• Recursion down to NB
  • Unfold completely below NB to get a basic block
  • Micro-Kernel:
    • The basic block compiled with native compiler
    • Best performance for NB =12
    • Compiler unable to use registers
    • Unfolding reduces control overhead
      • limited by I-cache

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Control Structures

- Recursion down to NB
  - Unfold completely below NB to get a basic block
- Micro-Kernel
- Scalarize all array references in the basic block
- Compile with native compiler

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive

Micro-Kernel
- None / Compiler
- Scalarized / Compiler

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Control Structures

Outer Control Structure

Iterative

Recursive

Inner Control Structure

Statement

Recursive

Micro-Kernel

None / Compiler

Scalarized / Compiler

Belady / BRILA

• Recursion down to NB
• Unfold completely below NB to get a basic block
  • Micro-Kernel
• Perform Belady’s register allocation on the basic block
  • Schedule using BRILA compiler

9/10:

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Control Structures

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive

Micro-Kernel
- None / Compiler
- Scalarized / Compiler
- Belady / BRILA
- Coloring / BRILA

- Recursion down to NB
- Unfold completely below NB to get a basic block
  - Micro-Kernel
- Construct a preliminary schedule
- Perform Graph Coloring register allocation
- Schedule using BRILA compiler
Control Structures

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive
- Iterative

Micro-Kernel
- None / Compiler
- Scalarized / Compiler
- Belady / BRILA
- Coloring / BRILA

• Recursion down to MU x N x KU
  • Micro-Kernel
    • Completely unroll MU x N x KU triply nested loop
    • Construct a preliminary schedule
    • Perform Graph Coloring register allocation
    • Schedule using BRILA compiler
Control Structures

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive
- Iterative

Mini-Kernel
- Recursion down to NB
  - Mini-Kernel
  - NB x NB x NB triply nested loop
  - Tiling for L1 cache
  - Body is Micro-Kernel

Micro-Kernel
- None / Compiler
- Scalarized / Compiler
- Belady / BRILA
- Coloring / BRILA

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Control Structures

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive
- Iterative

Specialized code generator with search
- Mini-Kernel
- Micro-Kernel

None / Compiler
- Scalarized / Compiler
- Belady / BRILA
- Coloring / BRILA

ATLAS CGw/S
ATLAS Unleashed

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Control Structures

Outer Control Structure
- Iterative
- Recursive

Inner Control Structure
- Statement
- Recursive
- Iterative

Mini-Kernel

Micro-Kernel

ATLAS CGw/S
ATLAS Unleashed

Statement

Recursive

None / Compiler
Scalorized / Compiler
Belady / BRILA
Coloring / BRILA

Slide source: Roeder, Yotov, Pingali at Cornell/UTexas
Experience

- In practice, need to cut off recursion
- Implementing a high-performance Cache-Oblivious code is not easy
  - Careful attention to micro-kernel and mini-kernel is needed
- Using fully recursive approach with highly optimized recursive recursive micro-kernel, Pingali et al report that they never got more than 2/3 of peak.
- Issues with Cache Oblivious (recursive) approach
  - Recursive Micro-Kernels yield less performance than iterative ones using same scheduling techniques
  - Pre-fetching is needed to compete with best code: not well-understood in the context of CO codes
Basic Linear Algebra Subroutines (BLAS)

- Industry standard interface (evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - vector operations: dot product, saxpy ($y=\alpha^*x+y$), etc
    - $m=2*n$, $f=2*n$, $q \sim 1$ or less
  - BLAS2 (mid 1980s)
    - matrix-vector operations: matrix vector multiply, etc
    - $m=n^2$, $f=2*n^2$, $q\sim 2$, less overhead
    - somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - matrix-matrix operations: matrix matrix multiply, etc
    - $m \leq 3n^2$, $f=O(n^3)$, so $q=f/m$ can possibly be as large as $n$, so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK & ScaLAPACK)
  - See www.netlib.org/\{lapack,scalapack\}
  - More later in course
BLAS speeds on an IBM RS6000/590

Peak speed = 266 Mflops

BLAS 3 (n-by-n matrix matrix multiply) vs BLAS 2 (n-by-n matrix vector multiply) vs BLAS 1 (saxpy of n vectors)
Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops
  - $O(n^{2.81})$
- Consider a $2 \times 2$ matrix multiply, normally takes 8 multiplies, 4 adds
  - Strassen does it with 7 multiplies and 18 adds

Let $M = \begin{pmatrix} m11 & m12 \\ m21 & m22 \end{pmatrix}$, $A = \begin{pmatrix} a11 & a12 \\ a21 & a22 \end{pmatrix}$, $B = \begin{pmatrix} b11 & b12 \\ b21 & b22 \end{pmatrix}$

Let
- $p1 = (a12 - a22) \cdot (b21 + b22)$
- $p2 = (a11 + a22) \cdot (b11 + b22)$
- $p3 = (a11 - a21) \cdot (b11 + b12)$
- $p4 = (a11 + a12) \cdot b22$
- $p5 = a11 \cdot (b12 - b22)$
- $p6 = a22 \cdot (b21 - b11)$
- $p7 = (a21 + a22) \cdot b11$

Then
- $m11 = p1 + p2 - p4 + p6$
- $m21 = p6 + p7$
- $m22 = p2 - p3 + p5 - p7$

Extends to $n \times n$ by divide & conquer
Strassen
(continued)

\[ T(n) = \text{Cost of multiplying } \times n \times \times n \text{ matrices} \]
\[ = 7 \times T(n/2) + 18 \times (n/2)^2 \]
\[ = O(n \log_2 7) \]
\[ = O(n^{2.81}) \]

- Asymptotically faster
- Several times faster for large \( n \) in practice
  - Cross-over depends on machine
  - Available in several libraries
- “Tuning Strassen's Matrix Multiplication for Memory Efficiency”, M. S. Thottethodi, S. Chatterjee, and A. Lebeck, in Proceedings of Supercomputing '98
- Caveats
  - Needs more memory than standard algorithm
  - Can be less accurate because of roundoff error
Other Fast Matrix Multiplication Algorithms

- Current world’s record is $O(n^{2.376...})$ (Coppersmith & Winograd)
- Why does Hong/Kung theorem not apply?
- Possibility of $O(n^{2+\varepsilon})$ algorithm! (Cohn, Umans, Kleinberg, 2003)
- Fast methods (besides Strassen) may need unrealistically large $n$
Optimizing in Practice

- Tiling for registers
  - loop unrolling, use of named “register” variables
- Tiling for multiple levels of cache and TLB
- Exploiting fine-grained parallelism in processor
  - superscalar; pipelining
- Complicated compiler interactions
- Hard to do by hand (but you’ll try)
- Automatic optimization an active research area
  - BeBOP: bebop.cs.berkeley.edu/
  - PHiPAC: www.icsi.berkeley.edu/~bilmes/phipac
    in particular tr-98-035.ps.gz
  - ATLAS: www.netlib.org/atlas
Removing False Dependencies

- Using local variables, reorder operations to remove false dependencies

\[
\begin{align*}
    a[i] &= b[i] + c; \\
    a[i+1] &= b[i+1] \times d; \quad \text{false read-after-write hazard between } a[i] \text{ and } b[i+1] \\
\end{align*}
\]

\[
\begin{align*}
    \downarrow \\
    \begin{align*}
    \text{float } f1 &= b[i]; \\
    \text{float } f2 &= b[i+1]; \\
    a[i] &= f1 + c; \\
    a[i+1] &= f2 \times d; \\
    \end{align*}
\end{align*}
\]

With some compilers, you can declare a and b unaliased.
- Done via “restrict pointers,” compiler flag, or pragma)
Exploit Multiple Registers

- Reduce demands on memory bandwidth by pre-loading into local variables

```c
while( ... ) {
    *res++ = filter[0]*signal[0]
        + filter[1]*signal[1]
        + filter[2]*signal[2];
    signal++;
}
```

```c
float f0 = filter[0];
float f1 = filter[1];
float f2 = filter[2];
while( ... ) {
    *res++ = f0*signal[0]
        + f1*signal[1]
        + f2*signal[2];
    signal++;
}
```

Example is a convolution
Minimize Pointer Updates

- Replace pointer updates for strided memory addressing with constant array offsets

```c
f0 = *r8; r8 += 4;
f1 = *r8; r8 += 4;
f2 = *r8; r8 += 4;
```

Down

```c
f0 = r8[0];
f1 = r8[4];
f2 = r8[8];
r8 += 12;
```

Pointer vs. array expression costs may differ.
- Some compilers do a better job at analyzing one than the other
Loop Unrolling

- Expose instruction-level parallelism

```c
float f0 = filter[0], f1 = filter[1], f2 = filter[2];
float s0 = signal[0], s1 = signal[1], s2 = signal[2];
*res++ = f0*s0 + f1*s1 + f2*s2;
    do {
        signal += 3;
        s0 = signal[0];
        res[0] = f0*s1 + f1*s2 + f2*s0;

        s1 = signal[1];
        res[1] = f0*s2 + f1*s0 + f2*s1;

        s2 = signal[2];
        res[2] = f0*s0 + f1*s1 + f2*s2;
        res += 3;
    } while( ... );
```
Exposé Independent Operations

- **Hide instruction latency**
  - Use local variables to expose independent operations that can execute in parallel or in a pipelined fashion
  - Balance the instruction mix (what functional units are available?)

\[
\begin{align*}
  f_1 &= f_5 \times f_9; \\
  f_2 &= f_6 + f_{10}; \\
  f_3 &= f_7 \times f_{11}; \\
  f_4 &= f_8 + f_{12};
\end{align*}
\]
Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality

<table>
<thead>
<tr>
<th>Original matrix (numbers are addresses)</th>
<th>Reorganized into 2x2 blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 4 8 12</td>
<td>0 2 8 10</td>
</tr>
<tr>
<td>1 5 9 13</td>
<td>1 3 9 11</td>
</tr>
<tr>
<td>2 6 10 14</td>
<td>4 6 12 13</td>
</tr>
<tr>
<td>3 7 11 15</td>
<td>5 7 14 15</td>
</tr>
</tbody>
</table>
Locality in Other Algorithms

* The performance of any algorithm is limited by $q$.
* In matrix multiply, we increase $q$ by changing computation order.
  - increased temporal locality

* For other algorithms and data structures, even hand-transformations are still an open problem.
  - sparse matrices (reordering, blocking)
    - Weekly research meetings
    - Bebop.cs.berkeley.edu
    - About to release OSKI – tuning for sparse-matrix-vector multiply
  - trees (B-Trees are for the disk level of the hierarchy)
  - linked lists (some work done here)