Matrix Multiply Example

C-Level Code:

```c
int A[N][N], B[N][N], C[N][N];
for (int i=0; i < N; i++)
  for (j=0; j < N; j++) {
    C[i][j] = 0;
    for (k=0; k < N; k++)
      C[i][j] = C[i][j] + (A[i][k] * B[k][j]);
  }
```

Machine-Level:

```c
... Initialize pointers ...
L:
  if (inner loop done)  // Exit inner loop:
    R4 ← *R1;            // Load register 4 from memory
    R5 ← *R2;            // Load register 5 from memory
    R4 ← R5 * R4;        // Multiply
    R6 ← R6 + R4;        // Value of C[i][j]
    *R3 ← R6;            // Store result back to memory
    R1 = R1 + 4;         // Move pointer to next A[i][j]
    R2 = R2 + ...;       // Move pointer to next B[k][j]
  goto L;               // End of inner loop
```

Hardware Context

- Processor Status Word
  - Instruction I
  - Instruction I'
  - Status Register
- General Registers
  - GR[0]
  - ... GR[7]
- Heap
- Stack
- Stack Pointer
  - Save/restore hdw context when switching from one process to another

Topics

- Instruction execution
  - Registers
  - Pipelining
- CPU-Memory speed gap
- Cache memory
  - Temporal locality
  - Cache complexities (cache line-size, latency)
- Virtual memory
- Simple interrupts
- System call
Processor Registers

- **Program Counter (PC)**
  > Contains address of next instruction

- **Instruction Register (IR)**
  > Contains the most recently fetched instruction

- **Status Register (SR)**
  > Results of comparisons, errors, etc.
  > Sometimes called *Processor Status Word (PSW)*

- **Stack Pointer (SP)**
  > Address of the top stack element

- **General Registers (R[0]..R[7])**
  > Operands

Pentium Clock Speeds

- CPU Registers
- L1 Cache
- L2 Cache
- 250 MHz SRAM
- 500 MHz Pentium Processor Bus

- Video
- Bridge
- Memory
- 100 MHz SDRAM

- PCI Bus (32 or 64 bits), 33 MHz
- Need for asynchrony between components
  > Buffers smooth out traffic
  > May still get "stalls"

Evolution of Intel Processor Features

<table>
<thead>
<tr>
<th>Processor</th>
<th>Date</th>
<th>Frequency</th>
<th>Transistors</th>
<th>Caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>60 MHz</td>
<td>3.1 M</td>
<td>L1: 16KB</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>200 MHz</td>
<td>5.5 M</td>
<td>L1: 16 KB L2: 256 KB</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1997</td>
<td>266 MHz</td>
<td>7 M</td>
<td>L1: 32 KB L2: 256 KB</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>500 MHz</td>
<td>8.2 M</td>
<td>L1: 32 KB L2: 512 KB</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2000</td>
<td>1.5 GHz</td>
<td>42 M</td>
<td>*L1: 8 KB L2: 256 KB</td>
</tr>
<tr>
<td>Xeon</td>
<td>2002</td>
<td>1.70 GHz</td>
<td>42 M</td>
<td>*L1: 8 KB L2: 512 KB</td>
</tr>
<tr>
<td>Pentium M</td>
<td>2004</td>
<td>2.00 GHz</td>
<td>140 M</td>
<td>*L1: 64 KB L2: 2 MB</td>
</tr>
</tbody>
</table>

- * On-die caches
- 1 MHz = 10^6 cycles per sec, 1 GHz = 10^9

Pipelining Concept

- Sequential Process Example (4 operations)
  > op(0); op(1); op(2); op(3);
  > Time: t(0) + t(1) + t(2) + t(3)

- Pipelining Opportunity:
  > Apply process to stream of data: x[0], x[1], ...

- Ideal Pipeline of Example (t(i) = 1, all i)

  Time
  0 1 2 3 4 5 6 7

  Completion Time ?
  Completion Rate ?
  1 stage
Basic Instruction Execution

Start

Fetch Next Instruction

Register Inst.

Memory Inst.

Execute

Fetch Data

Execute

Done

Real processors have many more stages

Pipelining Instruction Execution

A Gantt Chart is a space-time diagram

CPU Clock

Tick

Fetch I

Fetch D

Execute

Ideal Case
Starting at tick 3, 1 instruction finishes every tick

CPU can “stall” for various reasons (memory wait)

Storage Hierarchy Properties

<table>
<thead>
<tr>
<th>* 1 KB = (2^{10}) = 1024 Bytes, 1 MB = (2^{20}), 1 GB = (2^{30})</th>
<th>Size</th>
<th>Access Time</th>
<th>Cost/MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Integer Registers</td>
<td>256 B</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Primary (L1) Cache (SRAM)</td>
<td>36 KB</td>
<td>1-4 ns</td>
<td>?</td>
</tr>
<tr>
<td>Secondary (L2) Cache</td>
<td>1 MB</td>
<td>4-30 ns</td>
<td>?</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
<td>512 MB</td>
<td>30-60 ns</td>
<td>$0.30</td>
</tr>
<tr>
<td>Disk Drive</td>
<td>160 GB</td>
<td>8-30 ms</td>
<td>$0.0008</td>
</tr>
</tbody>
</table>

Technology Trends

CPU speed doubles every 18 months (Moore’s Law)
Memory speed doubles every 10 years
But memory density quadruples every 2 years
Cache memories attempt to bridge CPU-memory gap
2-Level Cache Memory

- **Cache Memory**
  - Copy data to storage that is closer (and faster) to its place of use
  - Decrease access time on repeated accesses

- **Read Memory Operation**
  ```
  if (X is in cache) { // Hit
    Read X from cache; // T1
  } else { // Miss
    Read X from memory into cache; // T2
    Read X from cache; // + T1
  }
  ```

- **Cache design is complicated**

**Cache Memory Example**

- **Inner Loop**
  - For M=8: EMAT(data) = ?
  ```
  R x[0], - , W x[0], - , - , R x[M], - , - , W x[M], - , - ,...
  Miss Hit Miss Hit
  T2 + T1 T1 T2 + T1 T1
  ```

- **EMAT**
  - **Effective Memory Access Time**
  ```
  EMAT = [ H x T1 + M x (T1 + T2) ] / (H + M)
  ```
  - H: Number of cache hits
  - M: Number of cache misses
  - T1: Cache access time
  - T2: Memory access time
  - EMAT = T1 + m x T2 where m = (1-h)
  - m = 1-h: Miss ratio
  - Indicates you always pay the cost of a cache access
Temporal Locality

Consider the following program:

\[
S = 0; \\
\text{for } (i=0; i < N; i++) \{ \ S = S + x[i]; \} \\
T = 0; \\
\text{for } (i=0; i < N; i++) \{ \ T = T + x[i] \times x[(i\%4)\times100]; \}
\]

**High temporal locality**

- “A recently referenced memory location will be referenced again in the near future”
- Equivalently: The distance between references in the memory reference string is small

**Examples**

- Small inner loops of instructions that fit into cache memory
- Small enough \( N \)
- The variables \( x[0], x[100], x[200], x[300] \) are repeatedly accessed in the second loop; i.e., closely accessed in time

Memory System Complexities

- “Chunk” of cache (cache line)
  - Size may be different than size of main memory “chunk”; e.g.,
    - Memory accessed in 8-byte chunks
    - But cache line is 32 bytes ➔ Need 4 memory accesses
  - ➔ Multiple memory accesses to fill 1 cache line

- Main memory access latency
  - Must wait multiple memory cycles to access first chunk
  - May be able to get subsequent chunk in one memory cycle
  - e.g., 5-1-1-1 memory (8 memory cycles to get 32 bytes)

- Cache organization effects performance

Virtual Memory

- Map memory addresses at run-time
  - Physical (actual) address = \( f(\text{logical address}) \)
  - Typically, \( f(\) is implemented as a page table

- Basic Ideas
  - Hide details of real physical memory from user
  - Each user has \( n \) contiguous (linear) address spaces
    - Each begins at address 0
    - Paging (\( n = 1 \)) versus Segmentation (\( n \geq 1 \))

Virtual Memory (Paging)

Virtual Memory (Paging)

Virtual Address

<table>
<thead>
<tr>
<th>page#</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Address</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>frame#</th>
<th>offset</th>
</tr>
</thead>
</table>

Program

Paging Hardware

Memory

Valid Bit (In memory?)

Modified Bit (Dirty?)

Page Frame
Simple Interrupts (1)

- **Interrupt (Trap, Exception)**
  - A vectored transfer of control to the supervisor
  - Through a *trap table*
    - One entry for each trap type
    - Branch Table: 256 addresses stored in first 1KB of memory

- **Examples**
  - User requests OS service
    - *system call* via *software trap*
  - I/O device request completion
  - Arithmetic overflow or underflow
  - Page fault (virtual address not in main memory)
  - Memory-protection violation (segmentation fault)
  - Undefined instruction

Simple Interrupts (2)

- **Interrupt Processing**
  - Disable other interrupts while processing interrupt
  - Ignore new interrupt until after reenabling interrupts
  - Interrupts usually have an interrupt priority

Examples
- User requests OS service via system call
- I/O device request completion
- Arithmetic overflow or underflow
- Page fault (virtual address not in main memory)
- Memory-protection violation (segmentation fault)
- Undefined instruction

Life Of A System Call

- **User Process**
  - read (fd,buf,nb)
  - syscall(proc,arg, …)
  - C library functions
  - Top Half of Kernel (may block)
  - I/O Wait Queue (kernel mode)
  - User's buffer
  - File Descriptor, Buffer Address, Number of Bytes
  - Process State
  - CPU State

- **Bottom Half of Kernel (never blocks)**
  - Interrupt Handler
  - Hardware Interrupt
  - I/O Buffer
  - I/O controller buffer

Blocking Read System Call

- **User Program**
  - read (fd,buf,nb)
  - 1 Push nb onto stack
  - 2 Push buf onto stack
  - 3 Push fd onto stack
  - 4 Call read
  - 5 Reg = read op-code
  - 6 Trap to OS kernel
  - 7 Check syscall args
  - 8 Jump thru syscall table
  - 9 Exec syscall handler
  - 10 Scheduler
  - Enter kernel mode

- **OS Kernel**
  - 11 Interrupt handler

- **Contexts/Modes:**
  - User: Instr, Heap
  - Kernel: Instr, Stack
Life Of A Device Read Request (1)

- **CPU**
  - `read()` results in `syscall()` which traps into kernel
    - Machine code for `read()` code comes from a library
    - Enter *kernel mode* from *user mode* *(Perform mode switch)*
      - Allows privileged instructions and access to all of memory
  - Queue request if I/O device is not ready
  - Send read request to controller of I/O device
  - Put process on queue while waiting for I/O to finish
  - Perform *context switch*
    - Save process state and switch control (give CPU) to another process
- **Device Controller**
  - Initiate operation on I/O device

Life Of A Device Read Request (2)

- **I/O Device**
  - Transmit data to the controller's buffer
- **Device Controller**
  - Request use of bus
  - Transfer data to main memory after bus grant
    - Controller "shares" memory bus with CPU and other devices
  - *Interrupt CPU* when read request has finished
- **CPU**
  - Interrupt transfers CPU control to the interrupt service routine
  - Save state of interrupted process
  - Disable lower priority interrupts while handling interrupt
  - Give control of CPU to a process selected by the scheduler
    - Restore CPU state and copy data from kernel buffer to user buffer
  - Reenable interrupts, restore user-mode and return from syscall

Privileged Instructions

- **Examples**
  - Perform I/O
  - Change virtual memory protection bits
  - Disable/enable interrupt
- **Two (2) Modes of Operation**
  - *User Mode*: Execute instructions in user program
  - *Supervisor (Kernel) Mode*
    - Can execute privileged instructions
    - Can access VM pages in operating system kernel
  - *Mode bit* in *status register* indicates execution mode
  - Causes of mode switching
    - Interrupt, System call (software trap)
- **Execute privileged instr. only in kernel mode!!**

Reading

- Tanenbaum, Section 2.1
- Tanenbaum, Sections 10.1-10.3
  - Focus: `fork()`, `waitpid()`, `execv()` and other exec system calls