Optimizing Memory Bandwidth

Don't settle for just a byte or two.

Grab a whole fistful of cache.

Mike Wall
Member of Technical Staff
Developer Performance Team
Advanced Micro Devices, Inc.

PC performance in 2002: Fast CPUs waiting for slow memory

CPU speed is improving faster than memory *latency*

- Prefetch read data into the cache, to hide latency
- Write to in-cache buffers when possible

CPU speed is improving faster than memory *bandwidth*

→→ •Maximize data I/O speed for reading/writing cache blocks

- Reduce memory traffic by reordering operations:
    - Process data in blocks
    - Do ***all*** operations on one block, in the cache, then process the next block

Survey questions for the crowd

Who uses assembly language optimization? Do you use in-line assembly code, or separate .asm files?

Who writes system benchmark code that runs at start-up, calibrating different code paths and/or options?

Who does run-time CPU feature identification and has separate performance-critical code paths for different CPUs?

Who is developing their games to make scalable use of the CPU, i.e. better physics, more polys, better audio etc. on faster CPU?

Three examples:

Copying a block of memory

- A very simple, common operation
- Step by step example
- Illustrate main optimization methods

Adding arrays of type “double”

- Build upon the memcopy methods
- Perform operations on the data
- Use the X87 FPU along with MMX™ instructions

C++ source example

- Implement key methods w/o assembly code

Memory copy: step 1

rep movsb: baseline code ~620 MB/sec

reference machine: AMD Athlon^{TM}XP Processor 1800+ with DDR memory

```
mov   esi, [src]      // source array
mov   edi, [dst]      // destination array

mov   ecx, [len]      // number of QWORDS (8 bytes)
shl   ecx, 3          // convert to byte count

rep   movsb
```

Memory copy: step 2

rep movsd: copy DWORDs   ~640 MB/sec (3% faster)

```
mov   esi, [src]       // source array
mov   edi, [dst]       // destination array

mov   ecx, [len]       // number of QWORDS (8 bytes)
shl   ecx, 1           // convert to DWORD count

rep   movsd
```

GameDevelopers Conference

Memory copy: step 3

MOV: use “RISC” ops ~650 MB/sec (1.5% faster)

```
    mov  esi, [src]       // source array
    mov  edi, [dst]       // destination array

    mov  ecx, [len]       // number of QWORDS (8 bytes)
    shl  ecx, 1           // convert to DWORD count

copyloop:
    mov  eax, dword ptr [esi]
    mov  dword ptr [edi], eax
    add  esi, 4
    add  edi, 4
    dec  ecx
    jnz  copyloop
```

make
better
games

Memory copy: step 4

unroll: reduce loop overhead ~640 MB/sec (1.5% slower)

```
    mov   esi, [src]       // source array
    mov   edi, [dst]       // destination array

    mov   ecx, [len]       // number of QWORDS (8 bytes)
    shr   ecx, 1           // convert to 16-byte size count
                           // (assumes len / 16 is an integer)
copyloop:
    mov   eax, dword ptr [esi]
    mov   dword ptr [edi], eax
    mov   ebx, dword ptr [esi+4]
    mov   dword ptr [edi+4], ebx
    mov   eax, dword ptr [esi+8]
    mov   dword ptr [edi+8], eax
    mov   ebx, dword ptr [esi+12]
    mov   dword ptr [edi+12], ebx
    add   esi, 16
    add   edi, 16
    dec   ecx
    jnz   copyloop
```

GameDevelopers Conference

Memory copy: step 5

group: read/read, write/write ~660 MB/sec (3% faster)

```
    mov   esi, [src]      // source array
    mov   edi, [dst]      // destination array

    mov   ecx, [len]      // number of QWORDS (8 bytes)
    shr   ecx, 1          // convert to 16-byte size count

copyloop:
    mov   eax, dword ptr [esi]
    mov   ebx, dword ptr [esi+4]
    mov   dword ptr [edi], eax
    mov   dword ptr [edi+4], ebx
    mov   eax, dword ptr [esi+8]
    mov   ebx, dword ptr [esi+12]
    mov   dword ptr [edi+8], eax
    mov   dword ptr [edi+12], ebx
    add   esi, 16
    add   edi, 16
    dec   ecx
    jnz   copyloop
```

make better games

Memory copy: step 6

use MMX^{TM}: eight 64-bit regs ~705 MB/sec (7% faster)

```
    mov   esi, [src]      // source array
    mov   edi, [dst]      // destination array

    mov   ecx, [len]      // number of QWORDS (8 bytes)
                          // assumes len / 8 = integer

→   lea   esi, [esi+ecx*8]  // end of source
→   lea   edi, [edi+ecx*8]  // end of destination

→   neg   ecx               // use a negative offset
                            //  as a combo
                            //   pointer-and-loop-counter
```

(continued on next page)

GameDevelopers Conference

Memory copy: step 6 (cont.)

```
copyloop:
    movq  mm0, qword ptr [esi+ecx*8]
    movq  mm1, qword ptr [esi+ecx*8+8]
    movq  mm2, qword ptr [esi+ecx*8+16]
    movq  mm3, qword ptr [esi+ecx*8+24]
    movq  mm4, qword ptr [esi+ecx*8+32]
    movq  mm5, qword ptr [esi+ecx*8+40]
    movq  mm6, qword ptr [esi+ecx*8+48]
    movq  mm7, qword ptr [esi+ecx*8+56]

    movq  qword ptr [edi+ecx*8],    mm0
    movq  qword ptr [edi+ecx*8+8],  mm1
    movq  qword ptr [edi+ecx*8+16], mm2
    movq  qword ptr [edi+ecx*8+24], mm3
    movq  qword ptr [edi+ecx*8+32], mm4
    movq  qword ptr [edi+ecx*8+40], mm5
    movq  qword ptr [edi+ecx*8+48], mm6
    movq  qword ptr [edi+ecx*8+56], mm7

    add  ecx, 8
    jnz  copyloop
→   emms   // empty MMX state
```

make better games

GameDevelopers Conference

Memory copy: step 7

MOVNTQ: streaming store   ~1130 MB/sec (60% faster!)

```
mov  esi, [src]      // source array
mov  edi, [dst]      // destination array

mov  ecx, [len]      // number of QWORDS (8 bytes)
                     // assumes len / 8 = integer

lea  esi, [esi+ecx*8]  // end of source
lea  edi, [edi+ecx*8]  // end of destination

neg  ecx               // use a negative offset
                       //  as a combo
                       //   pointer-and-loop-counter
```

(continued on next page)

make better games

GameDevelopers Conference

Memory copy: step 7 (cont.)

```
copyloop:
    movq  mm0, qword ptr [esi+ecx*8]
    movq  mm1, qword ptr [esi+ecx*8+8]
    movq  mm2, qword ptr [esi+ecx*8+16]
    movq  mm3, qword ptr [esi+ecx*8+24]
    movq  mm4, qword ptr [esi+ecx*8+32]
    movq  mm5, qword ptr [esi+ecx*8+40]
    movq  mm6, qword ptr [esi+ecx*8+48]
    movq  mm7, qword ptr [esi+ecx*8+56]

→   movntq  qword ptr [edi+ecx*8],    mm0
→   movntq  qword ptr [edi+ecx*8+8],  mm1
→   movntq  qword ptr [edi+ecx*8+16], mm2
→   movntq  qword ptr [edi+ecx*8+24], mm3
→   movntq  qword ptr [edi+ecx*8+32], mm4
→   movntq  qword ptr [edi+ecx*8+40], mm5
→   movntq  qword ptr [edi+ecx*8+48], mm6
→   movntq  qword ptr [edi+ecx*8+56], mm7

    add  ecx, 8
    jnz  copyloop
→   sfence  // flush write buffer
    emms
```

make better games

GameDevelopers Conference

Memory copy: step 7 (cont.)

Why does MOVNTQ write so much faster?

•Write data bypasses the cache, and goes directly to a write combining buffer, so…

•***No cache line allocation = no data read from memory!***

•Eliminates ½ the memory traffic for write operations.

•Use when you don't need to access the data again soon.

make better games

GameDevelopers Conference

Memory copy: step 8

PREFETCH: look ahead ~1240 MB/sec (10% faster)

```
mov  esi, [src]      // source array
mov  edi, [dst]      // destination array

mov  ecx, [len]      // number of QWORDS (8 bytes)
                     // assumes len / 8 = integer

lea  esi, [esi+ecx*8]  // end of source
lea  edi, [edi+ecx*8]  // end of destination

neg  ecx               // use a negative offset
                       //  as a combo
                       //   pointer-and-loop-counter
```

(continued on next page)

make better games

Memory copy: step 8 (cont.)

```
copyloop:
→   prefetchnta  [esi+ecx*8 + 512]   // fetch ahead by 512 bytes

    movq  mm0, qword ptr [esi+ecx*8]
    movq  mm1, qword ptr [esi+ecx*8+8]
    movq  mm2, qword ptr [esi+ecx*8+16]
    movq  mm3, qword ptr [esi+ecx*8+24]
    movq  mm4, qword ptr [esi+ecx*8+32]
    movq  mm5, qword ptr [esi+ecx*8+40]
    movq  mm6, qword ptr [esi+ecx*8+48]
    movq  mm7, qword ptr [esi+ecx*8+56]
    movntq  qword ptr [edi+ecx*8],    mm0
    movntq  qword ptr [edi+ecx*8+8],  mm1
    movntq  qword ptr [edi+ecx*8+16], mm2
    movntq  qword ptr [edi+ecx*8+24], mm3
    movntq  qword ptr [edi+ecx*8+32], mm4
    movntq  qword ptr [edi+ecx*8+40], mm5
    movntq  qword ptr [edi+ecx*8+48], mm6
    movntq  qword ptr [edi+ecx*8+56], mm7
    add  ecx, 8
    jnz  copyloop
    sfence
    emms
```

Memory copy: step 8 (cont.)

Why does PREFETCHNTA improve performance?

•The CPU starts to fetch the data sooner, before it's actually needed, and this hides some of the memory latency.

•There is also an automatic hardware prefetcher, which detects sequential memory access patterns, but the software PREFETCHNTA instruction can still help performance in highly optimized code.

Memory copy: step 9

Block Prefetch: grab a whole fistful of cache
~1976 MB/sec (59% faster, up 300% vs. original!)

```
→ #define CACHEBLOCK 400h   // number of QWORDs in a chunk (8K bytes)

    mov  esi, [src]      // source array
    mov  edi, [dst]      // destination array

    mov  ecx, [len]      // number of QWORDS (8 bytes)
                         // assumes len / CACHEBLOCK is an integer

    lea  esi, [esi+ecx*8]  // end of source
    lea  edi, [edi+ecx*8]  // end of destination

    neg  ecx               // use a negative offset
                           //  as a combo
                           //   pointer-and-loop-counter
```

(continued on next page)

Memory copy: step 9 (cont.)

```
mainloop:

    mov   eax, CACHEBLOCK / 16    // note: prefetch loop is unrolled 2X
    add   ecx, CACHEBLOCK         // move up to end of block
prefetchloop:
→    mov   ebx, [esi+ecx*8-64]   // read one address in this cache line...
→    mov   ebx, [esi+ecx*8-128] //   ... and one in the previous line
    sub   ecx, 16               // 16 QWORDS = 2 64-byte cache lines
    dec   eax
    jnz   prefetchloop
```

(continued on next page)

Memory copy: step 9 (cont.)

```
    mov   eax, CACHEBLOCK / 8
writeloop:
    movq  mm0, qword ptr [esi+ecx*8]      // reads from cache!
    .
    .
    movq  mm7, qword ptr [esi+ecx*8+56]

    movntq  qword ptr [edi+ecx*8],    mm0  // streaming store
    .
    .
    movntq  qword ptr [edi+ecx*8+56], mm7

    add   ecx, 8
    dec   eax
    jnz   writeloop

    or    ecx, ecx      // assumes integer number of cacheblocks
    jnz   mainloop

    sfence
    emms
```

Memory copy summary

Why does Block Prefetch improve performance?

•Reading just ***one*** address per cache line brings the entire line into cache, with the smallest number of read requests.

•Using the MOV instruction forces all lines to be read, whereas the software PREFETCHNTA can sometimes be ignored (MOV can't be ignored).

•Reading a large block, then writing a large block, causes the smallest number of read/write mode changes in the memory module.

•Reading in reverse order can sometimes run a bit faster.

Example 2: Floating point array addition

c[x] = a[x] + b[x]

- Build on the memory copy optimizations
- Use X87 floating point unit for the addition
- Deal with mixing MMX™ and X87 code

GameDevelopers Conference

Floating point array addition

Unrolled loop: baseline performance ~950 MB/sec

reference machine: AMD Athlon™XP Processor 1800+ with DDR memory

```
mov  esi, [src1]      // source array one
mov  ebx, [src2]      // source array two
mov  edi, [dst]       // destination array

mov  ecx, [len]       // number of Doubles (8 bytes)
                      // assumes len / 8 = integer

lea  esi, [esi+ecx*8] // end of source #1
lea  ebx, [ebx+ecx*8] // end of source #2
lea  edi, [edi+ecx*8] // end of destination

neg  ecx               // use a negative offset
                       //  as a combo
                       //   pointer-and-loop-counter
```

(continued on next page)

make better games

GameDevelopers Conference

FPU array addition (baseline cont.)

```
addloop:

    fld  qword ptr [esi+ecx*8+56]
    fadd qword ptr [ebx+ecx*8+56]
    .
    .
    fld  qword ptr [esi+ecx*8+0]
    fadd qword ptr [ebx+ecx*8+0]

    fstp qword ptr [edi+ecx*8+0]
    .
    .
    fstp qword ptr [edi+ecx*8+56]

    add  ecx, 8
    jnz  addloop
```

FPU array addition (three phase)

Three phase processing ~1720 MB/sec (80% faster!)

```
→  #define CACHEBLOCK 400h   // number of QWORDs in a chunk (8K bytes)
→  int* storedest
→  char buffer[CACHEBLOCK * 8]  // in-cache temporary storage

    mov  esi, [src1]      // source array one
    mov  ebx, [src2]      // source array two
    mov  edi, [dst]       // destination array

    mov  ecx, [len]       // number of Doubles (8 bytes)
                          // (assumes len / CACHEBLOCK = integer)
    lea  esi, [esi+ecx*8]
    lea  ebx, [ebx+ecx*8]
    lea  edi, [edi+ecx*8]

→  mov  [storedest], edi // save the real dest for later

→  mov  edi, [buffer]    // temporary in-cache buffer...
→  lea  edi, [edi+ecx*8] // ... stays in cache from heavy use

    neg  ecx
```

(continued on next page)

FPU array add (phase 1: block prefetch)

```
mainloop:   // prefetch each block separately for best performance

    mov   eax, CACHEBLOCK / 16
    add   ecx, CACHEBLOCK
prefetchloop1:                      // block prefetch array #1
    mov   edx, [esi+ecx*8-64]
    mov   edx, [esi+ecx*8-128]       // (this loop is unrolled 2X)
    sub   ecx, 16
    dec   eax
    jnz   prefetchloop1

    mov   eax, CACHEBLOCK / 16
    add   ecx, CACHEBLOCK
prefetchloop2:                      // block prefetch array #2
    mov   edx, [ebx+ecx*8-64]
    mov   edx, [ebx+ecx*8-128]       // (this loop is unrolled 2X)
    sub   ecx, 16
    dec   eax
    jnz   prefetchloop2
```

(continued on next page)

FPU array add (phase 2: computation)

```
    mov   eax, CACHEBLOCK / 8
processloop:            // this loop read/writes all in cache!
    fld   qword ptr [esi+ecx*8+56]
    fadd qword ptr [ebx+ecx*8+56]
    .
    .
    fld   qword ptr [esi+ecx*8+0]
    fadd qword ptr [ebx+ecx*8+0]

    fstp qword ptr [edi+ecx*8+0]
    .
    .
    fstp qword ptr [edi+ecx*8+56]

    add   ecx, 8
    dec   eax
    jnz   processloop
```

(continued on next page)

GameDevelopers Conference

FPU array addition (phase 3: write back)

```
        sub   ecx, CACHEBLOCK
        mov   edx, [storedest]
        mov   eax, CACHEBLOCK / 8
    writeloop:                             // write buffer to main mem
        movq   mm0, qword ptr [edi+ecx*8+0]
        movq   mm1, qword ptr [edi+ecx*8+8]
        movq   mm2, qword ptr [edi+ecx*8+16]
        movq   mm3, qword ptr [edi+ecx*8+24]
        movq   mm4, qword ptr [edi+ecx*8+32]
        movq   mm5, qword ptr [edi+ecx*8+40]
        movq   mm6, qword ptr [edi+ecx*8+48]
        movq   mm7, qword ptr [edi+ecx*8+56]
        movntq   qword ptr [edx+ecx*8+0],   mm0
        movntq   qword ptr [edx+ecx*8+8],   mm1
        movntq   qword ptr [edx+ecx*8+16], mm2
        movntq   qword ptr [edx+ecx*8+24], mm3
        movntq   qword ptr [edx+ecx*8+32], mm4
        movntq   qword ptr [edx+ecx*8+40], mm5
        movntq   qword ptr [edx+ecx*8+48], mm6
        movntq   qword ptr [edx+ecx*8+56], mm7
        add   ecx, 8
        dec   eax
        jnz   writeloop
```

FPU array addition (end loops)

```
    or   ecx, ecx
    jge  exit

    sub  edi, CACHEBLOCK * 8    // reset edi back to start of buffer

    sfence                      // flush the write buffer when done
    emms                        // empty the MMX state

    jmp  mainloop               // iterate the three phase processing

exit:
    sfence
    emms
```

FPU array addition (summary)

Why does three phase processing improve performance?

- Maximum possible memory bandwidth is achieved during phase 1 (block prefetch) and phase 3 (write back).

- All of phase 2 (computation) reads and writes within cache, so it executes at maximum possible speed.

- Caveat: does not allow overlap of memory access and computation, so the technique offers most benefit when phase 2 (computation) is simple and fast.

C++ source code example

Even if you can't use assembly language, you can still implement block prefetch to improve read performance.

Example: add floating point values from 2 arrays

$$c = \sum a[x] + \sum b[x]$$

Baseline C++ loop gets about 1000 MB/sec on reference machine: AMD Athlon™XP Processor 1800+ with DDR memory

```
for (int i = 0; i < MEM_SIZE; i += 8) {  // 8 bytes per double
   double summo += *a_ptr++ + *b_ptr++;  // Reads from memory
}
```

GameDevelopers Conference

C++ source code example (cont.)

The same function, but optimized to read chunks of the arrays into cache at maximum bandwidth.

First, the routine that performs the block prefetch in source code:

```
int p_fetch;  // this global "anchor" variable helps to fool the compiler’s
              //  optimizer into leaving the prefetch code intact!

static const void inline BLOCK_PREFETCH_4K(void* addr) {
    int* a = (int*) addr;    // cast as INT pointer for speed

  //<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
    p_fetch +=  a[0] +  a[16] +  a[32] +  a[48]  // Grab every
            +  a[64] +  a[80] +  a[96] + a[112]  //  64th address,
            + a[128] + a[144] + a[160] + a[176]  //   to hit each
            + a[192] + a[208] + a[224] + a[240]; //    cache line once.
    a += 256;   // advance to next 1K block
  //<><><> repeat 3 more times, for a total of 4K bytes fetched <><><><>

}
```

make better games

C++ source code example (cont.)

Now the main loop, modified to call the Block Prefetch routine:

```
static const int CACHEBLOCK = 0x1000; // prefetch block size (4K bytes)

for (int m = 0; m < MEM_SIZE; m += CACHEBLOCK) {  // process in blocks

        BLOCK_PREFETCH_4K(a_ptr);  // read 4K bytes of “a” into cache
        BLOCK_PREFETCH_4K(b_ptr);  // read 4K bytes of “b” into cache

        for (int i = 0; i < CACHEBLOCK; i += 8) {
           double summo += *a_ptr++ + *b_ptr++;  // Reads from cache!
        }
}
```

This version gets over 1430 MB/sec, more than 40% faster!

Tools and techniques

•Profile and find hot spots!   Measure, don't assume.   You can use the free profiler in AMD CodeAnalyst (on AMD web site)

•Build a simple test case so you can easily tweak, measure, tweak, measure etc. until you get the best results

•Test on all your target platforms to validate results

Fast code looks like…

•Optimize at the algorithm level! Organize data for speed, align it, process in-cache blocks, in-line small functions. Use all the source-level performance guidelines (see AMD Athlon^{TM} Optimization Guide)

•But ultimately, there’s no substitute for assembly language

•Use MMX^{TM}, MOVNTQ and PREFETCHNTA. Every modern CPU has them, they’re easy to use, and they can really help performance.

•“Blended code” for all modern CPUs can run quite fast

•For absolute ultimate performance, custom code paths for different CPU families may be necessary

Take-away messages

•Optimization can really help performance!

•Memory bandwidth can really be a bottleneck, but…

•***Memory bandwidth can be optimized dramatically***

•Block prefetch, streaming stores (MOVNTQ), and three phase processing are good to have in your programmer's bag-o-tricks

Final word

Optimi$ation + $calability = $$

Optimization
- Provides better gaming at <u>every</u> PC performance level

Scalability
- increases your available market on the low end
- provides more excitement on the high end
- games remain “cool” longer, for longer viable shelf life

The hardware will keep improving rapidly, so aim high!