Using Block Prefetch for Optimized Memory Performance

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

Introduction

In recent years, clock speeds of X86 processors have risen rapidly. Also, the latest processors can execute several instructions in a single clock cycle. Without a doubt, today's mainstream CPUs have horsepower that was unattainable just a short time ago.

The main performance bottleneck, for many applications, is no longer processor speed. It is memory bandwidth.

New DRAM memory technologies, such as DDR, have provided an increase in raw memory bandwidth. How can applications take best advantage of this increase? One answer is described in this document. It involves two related ideas: block prefetch, and three phase processing.

Block prefetch is a technique for reading blocks of data from main memory at very high data rates. Three phase processing is a programming style that divides data into chunks which are read into cache using block prefetch, then operated on within the cache, then results are written out to memory, all with high efficiency.

The techniques are relevant to applications that access large, localized data objects in system memory, in a sequential or near-sequential manner. The basis of the techniques is to take best advantage of the processor's cache memory.

This document will start with the most basic and useful memory function: copying a block of data from one area of memory to another. This will be the vehicle for exploring the main optimization ideas. Then the ideas will be applied to optimizing a bandwidth-limited function that uses the FPU to process linear data arrays.

Note: These code samples were run on an AMD Athlon™XP Processor 1800+ with CAS2 DDR2100 memory, and VIA KT266A chipset. Data sizes were several megabytes, i.e. much larger than the cache. Exact performance numbers will be different on other platforms, but the basic techniques can be widely applicable across the spectrum of current PCs. As with any optimization project, be sure to benchmark the code on all relevant platforms, try different code variations to see what works best, and confirm your results.

The simplest way to copy memory is to use the REP MOVSB instruction. This is the automatic instruction provided by X86 for memory copy.

bandwidth: ~620 MB/sec (baseline)

```
    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
```

That's the starting point. An obvious improvement to try is increasing the byte copy to a 4-byte copy. REP MOVSD does the job.

bandwidth: ~640 MB/sec improvement: 3%

```
    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
```

That was a little improvement. Now, on modern processors, the REP MOVS instructions may not be as efficient as an explicit loop which uses simpler "RISC" instructions. The simple instructions can be executed in parallel, and sometimes even out-of-order, within the CPU. So next up is a loop which performs the copy by using MOV instructions.

bandwidth: ~650 MB/sec improvement: 1.5%

```
    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
```

The explicit loop was a tiny bit faster than REP MOVSD. Building on that explicit loop is the next optimization: unrolling the loop. This reduces the overhead of incrementing the pointers and counter, and reduces branching. Also, the [Register + Offset] form of addressing is used, which runs just as fast as the simple [Register] address. This is an unroll factor of 4X:

bandwidth: ~640 MB/sec improvement: -1.5%

```
    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
```

It's not clear why the performance dropped a little when the loop was unrolled, but there's good news. The unrolled loop presents an opportunity to apply another optimization trick: grouping reads together, and writes together. This can make the CPU's job easier, by enabling it to combine small cache data transfers into larger ones.

bandwidth: ~660 MB/sec improvement: 3%

```
    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
```

Grouping the read and write operations seemed to help. Grouping can be taken one step more. The MMX extensions, available on all modern X86 processors, provide eight large 64-bit registers that are ideal for this purpose. They permit 64 bytes of sequential reading, followed by 64 bytes of sequential writing.

This example also introduces a biased loop counter, which starts negative and counts up to zero; this allows the counter to serve double duty as a pointer, and eliminates the need for a CMP instruction to terminate the loop.

bandwidth: ~705 MB/sec improvement: 7%

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

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

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

    neg  ecx               // use a negative offset

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 the MMX state
```

Now that the MMX™ registers are being used, the code can employ a very special instruction: MOVNTQ. This is a streaming store instruction, for writing data to memory. This instruction bypasses the on-chip cache, and sends data directly into a write combining buffer. And because the MOVNTQ allows the CPU to avoid reading the old data from the memory destination address, MOVNTQ can effectively double the total write bandwidth. (note that an SFENCE is required after the data is written, to flush the write buffer)

bandwidth: ~1130 MB/sec improvement: 60% !!

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

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

    lea  esi, [esi+ecx*8]
    lea  edi, [edi+ecx*8]

    neg  ecx

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
    emms
```

The MOVNTQ instruction dramatically improved the speed of writing the data. Next is a faster way to read the data: a prefetch instruction. Prefetch can't increase the total read bandwidth, but it can get the processor started on reading the data before it's actually needed. If prefetch does its job, then the data will already be sitting in the cache when it's actually needed.

The automatic data prefetch function, sometimes called "hardware prefetch", can help also. In fact, it's already helping the performance here. Hardware prefetch detects sequential ascending memory access patterns, and automatically initiates a read request for the next cache line. But in highly optimized code like this example, a carefully adjusted software prefetch instruction can often improve read bandwidth even more than the hardware prefetch alone.

bandwidth: ~1240 MB/sec improvement: 10% !

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

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

    lea  esi, [esi+ecx*8]
    lea  edi, [edi+ecx*8]

    neg  ecx

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
```

Prefetch is clearly helping. At this point, a subtle but important variation on the theme can be used. It is called Block Prefetch. In previous examples, grouping reads together gave a boost to performance. Block prefetch is an extreme extension of this idea. The strategy is to read a large stream of sequential data into the cache, without any interruptions.

Significantly, the MOV instruction is used, rather than the software prefetch instruction. Unlike a prefetch instruction, which is officially only a "hint" to the processor, a MOV instruction cannot be ignored and must be executed in-order. The result is that the memory system reads sequential, back-to-back address blocks, which yields the fastest memory read bandwidth.

Because the processor always loads an entire cache line (e.g. 64 bytes) whenever it accesses main memory, the prefetch loop only needs to read ONE address per cache line. Reading just one address per cache line reduces pressure on the internal load/store queue, and is an important part of achieving maximum read performance in this extreme case.

One additional trick is to read the cache lines in descending address order, rather than ascending order. This can improve performance a bit, by keeping the processor's hardware prefetcher from issuing any redundant read requests.

bandwidth: ~1976 MB/sec improvement: 59% ! (up 300% vs. baseline)

```
#define CACHEBLOCK 400h   // number of QWORDs in a chunk

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

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

    lea  esi, [esi+ecx*8]
    lea  edi, [edi+ecx*8]

    neg  ecx

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

    mov  eax, CACHEBLOCK / 8
writeloop:
    movq  mm0, qword ptr [esi+ecx*8]
    movq  mm1, qword ptr [esi+ecx*8+8]
    movq  mm2, qword ptr [esi+ecx*8+16]
```

(code continues on next page...)


```
    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
    dec  eax
    jnz  writeloop
    or   ecx, ecx
    jnz  mainloop

    sfence
    emms
```

This code, using block prefetch and the MOVNTQ streaming store, achieves an overall memory bandwidth of 1976 MB/sec, which is over 90% of the theoretical maximum possible with DDR2100 memory.

That’s the end of the memory copy optimization study. It is clear that block prefetch is a valuable optimization trick for boosting memory read bandwidth. The next optimization study applies this trick to code that does more than simply copy data.

This next optimization series applies the techniques just explored, in code that processes large arrays. An example which uses the X87 floating point unit is desirable, to illustrate the method for switching between MMX mode (which is needed for MOVNTQ store operations) and X87 FPU mode (which is used for the floating point calculations).

This example shows how to add two arrays of double precision floating point numbers together using the X87 FPU, and write the results to a third array.

As a baseline, a slightly optimized “unrolled loop” version would be something like this.

```
  bandwidth: ~950 MB/sec     (baseline performance)

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

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

    lea  esi, [esi+ecx*8]
    lea  ebx, [ebx+ecx*8]
    lea  edi, [edi+ecx*8]

    neg  ecx

addloop:

    fld  qword ptr [esi+ecx*8+56]
    fadd qword ptr [ebx+ecx*8+56]
    fld  qword ptr [esi+ecx*8+48]
    fadd qword ptr [ebx+ecx*8+48]
    fld  qword ptr [esi+ecx*8+40]
    fadd qword ptr [ebx+ecx*8+40]
    fld  qword ptr [esi+ecx*8+32]
    fadd qword ptr [ebx+ecx*8+32]
    fld  qword ptr [esi+ecx*8+24]
    fadd qword ptr [ebx+ecx*8+24]
    fld  qword ptr [esi+ecx*8+16]
    fadd qword ptr [ebx+ecx*8+16]
    fld  qword ptr [esi+ecx*8+8]
    fadd qword ptr [ebx+ecx*8+8]
    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+8]
    fstp qword ptr [edi+ecx*8+16]
    fstp qword ptr [edi+ecx*8+24]
    fstp qword ptr [edi+ecx*8+32]
    fstp qword ptr [edi+ecx*8+40]
    fstp qword ptr [edi+ecx*8+48]
    fstp qword ptr [edi+ecx*8+56]

    add  ecx, 8
    jnz  addloop
```

Now skipping ahead to a fully optimized version, the code appears as shown. The data is processed in blocks, as in the memory copy. But to completely implement the fast data copy techniques and also use X87 FPU mode, the processing takes place in three distinct phases: block prefetch, calculation, and memory write.

Why not two phases? Why not block prefetch the data in phase 1, then process and write the data in phase 2? The reason is MOVNTQ. In order to use MOVNTQ, the MMX registers must be used. Inside the processor, the MMX registers are mapped onto the X87 FPU registers, and switching between MMX mode and X87 FPU mode involves some time overhead. So for maximum performance the X87 FPU operations are done in phase 2, writing the results to an in-cache buffer. Then a third phase switches to MMX mode and uses MOVNTQ to copy the in-cache buffer out to main memory.

This general technique is called three phase processing. If only a small amount of phase 2 X87 FPU processing is needed, as in this particular example, the technique provides performance that approaches the optimized memory copy. As more FPU processing is required, memory bandwidth becomes less of a limiting factor, and the technique offers less advantage. So it must be used with discretion.

```
  bandwidth: ~1720 MB/sec    improvement: 80% !

#define CACHEBLOCK 400h   // number of QWORDs in a chunk
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 Floats (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

mainloop:

    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
```

(code continues on next page...)


```
    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

    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+48]
    fadd qword ptr [ebx+ecx*8+48]
    fld  qword ptr [esi+ecx*8+40]
    fadd qword ptr [ebx+ecx*8+40]
    fld  qword ptr [esi+ecx*8+32]
    fadd qword ptr [ebx+ecx*8+32]
    fld  qword ptr [esi+ecx*8+24]
    fadd qword ptr [ebx+ecx*8+24]
    fld  qword ptr [esi+ecx*8+16]
    fadd qword ptr [ebx+ecx*8+16]
    fld  qword ptr [esi+ecx*8+8]
    fadd qword ptr [ebx+ecx*8+8]
    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+8]
    fstp qword ptr [edi+ecx*8+16]
    fstp qword ptr [edi+ecx*8+24]
    fstp qword ptr [edi+ecx*8+32]
    fstp qword ptr [edi+ecx*8+40]
    fstp qword ptr [edi+ecx*8+48]
    fstp qword ptr [edi+ecx*8+56]

    add  ecx, 8
    dec  eax
    jnz  processloop

    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
(code continues on next page...)
```

```
    add  ecx, 8
    dec  eax
    jnz  writeloop

    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

exit:
    sfence
    emms
```

Summary:

Block prefetch and three phase processing are general techniques for improving the performance of memory-intensive applications on PCs. In a nutshell, the key points are:

#1 To get the maximum memory read bandwidth, read data into the cache in large blocks (e.g. 1K to 8K bytes), using block prefetch. When creating a block prefetch loop:

- -- unroll the loop by at least 2X
- -- use the MOV instruction (not the Prefetch instruction)
- -- read only one address per cache line
- -- read data into an ALU scratch register, like EAX
- -- read only one linear stream per loop
- -- to prefetch several streams, use a separate loop for each
- -- read cache lines in descending address order
- -- make sure all data is aligned

#2 To get maximum memory write bandwidth, write data from the cache to main memory in large blocks, using streaming store instructions. When creating a memory write loop:

- -- use the MMX registers to pass the data
- -- read from cache
- -- use MOVNTQ for writing to memory
- -- make sure the data is aligned
- -- write every address, in ascending order, without any gaps
- -- end with an SFENCE to flush the write buffer

#3 Whenever possible, code that actually “does the real work” should be reading its data from cache, and writing its output to an in-cache buffer. To enable this to happen, use #1 and #2 above.

Implementation detail note:

>>> Aligning “hot” branch targets to 16 byte boundaries can improve speed, by maximizing the number of instruction fills into the instruction-byte queue. This is especially important for short loops, like a block prefetch loop. This wasn’t shown in the code examples, for the sake of readability. It can be done with the ALIGN pragma, like this:

```
        align 16
        prefetchloop:
           mov  ebx, [esi+ecx*8-64]
           mov  ebx, [esi+ecx*8-128]
           sub  ecx, 16
           dec  eax
           jnz  prefetchloop
```

And remember: always benchmark your results, tweak until you get the best performance, and test on all relevant platforms.

Good luck optimizing your code!

Appendix A. C++ Source Code Implementation of Block Prefetch

The block prefetch technique can be applied at the source code level. This example uses C++, but it can be done in C or Fortran as well.

This code adds all the values in two large arrays of double precision floating point numbers, to produce a double precision floating point total.

Here is the ordinary, baseline C++ loop that does the job. It gets about 1000 MB/sec on an AMD Athlon™XP Processor 1800+ DDR machine:

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

Now here is the same function, but optimized using Block Prefetch to read the arrays into cache at maximum bandwidth. This Block Prefetch code reads 4 Kbytes of data at a time. This version achieves over 1430 MB/sec on the same machine, more than a 40% improvement!:

```
static const int CACHEBLOCK = 0x1000; // prefetch block size (4K bytes)
int p_fetch;                  // this "anchor" variable helps to
                              //   fool the compiler's optimizer

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
    p_fetch +=  a[0] +  a[16] +  a[32] +  a[48]
            +  a[64] +  a[80] +  a[96] + a[112]
            + a[128] + a[144] + a[160] + a[176]
            + a[192] + a[208] + a[224] + a[240];
    a += 256;
    p_fetch +=  a[0] +  a[16] +  a[32] +  a[48]
            +  a[64] +  a[80] +  a[96] + a[112]
            + a[128] + a[144] + a[160] + a[176]
            + a[192] + a[208] + a[224] + a[240];
    a += 256;
    p_fetch +=  a[0] +  a[16] +  a[32] +  a[48]
            +  a[64] +  a[80] +  a[96] + a[112]
            + a[128] + a[144] + a[160] + a[176]
            + a[192] + a[208] + a[224] + a[240];
}

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!
      }
}
```

Caution: Since the source-level prefetch code doesn't really "do" anything from the compiler's point of view, there is a danger that it might be entirely "optimized out" from the object code that is generated! So the block prefetch function BLOCK_PREFETCH_4K uses a trick to prevent that from happening. One value per cache line is read as an INT to prefetch the data, the INTs added together (which is very fast), and then they are assigned to a global variable. Now, the integer sum isn't really needed by the application, but this assignment should "fool" the compiler's optimizer into actually compiling the prefetch code intact. However, be aware that in general, compilers might try to remove block prefetch code unless care is taken.