有关公司采用 SIMD 指令的报告

一，什么是 SIMD 指令

SIMD 即单指令多数据操作，一般用于大型计算机的并行计算。

二，Intel 的 SIMD 技术

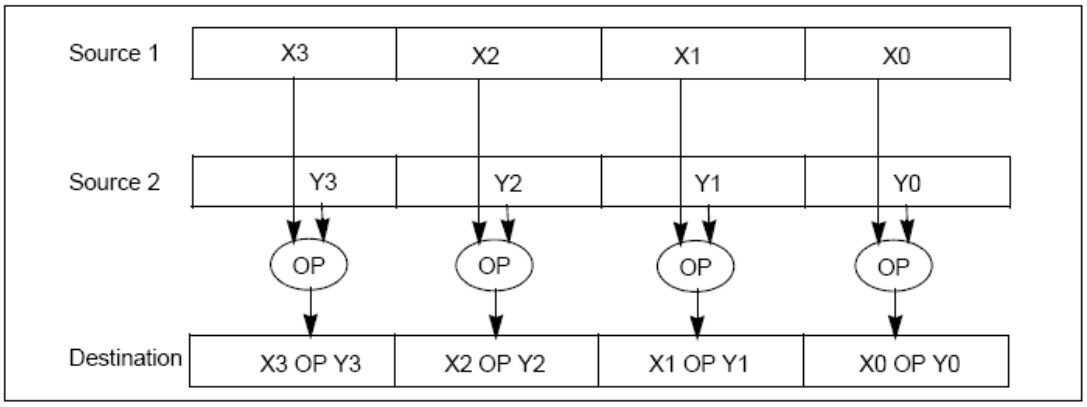
Intel 在 1993 年针对多媒体处理中大量反复的计算过程，将 SIMD 的思路引入了其处理器，采用特殊指令，实现了采用一条指令在一个时钟周期内对多个数据进行处理，从 Pentium 处理器开始到目前为止，共提出了四个扩展指令集：MMX，SSE，SSE2，SSE3。Intel 的 Hyper-Thread（HT）技术也属于 SIMD 技术的组成部分。各个扩展指令集不仅提供了对图像处理，视频处理，音频处理，通信处理和信号处理运算进行并行计算的指令，减少了计算所使用的时间，还提供了使用户可以直接控制 Cache 操作的指令，从而在消除 CPU 与内存之间通信的瓶颈上有着明显的功效。

MMX 指令集

MMX 指令集在 Pentium 和 PentiumII 处理器中引入，主要针对多媒体流处理运算，对 MMX 寄存器中的 Packed interger 进行操作，这种处理可以有效的用于整数数组的运算。

MMX 技术没有增加新的寄存器，而是采用了 x87FPU 的浮点寄存器（ST0，…ST7）（80Bit）的低 64 位，并将其成为 MMX 寄存器，在相应的时钟周期内对 MMX 寄存器中数据进行各种运算，在进行 Content Switch 时，CPU 将 MMX 寄存器等同浮点寄存器进行状态保存和恢复。

MMX 的核心内容就是并行计算，以 WORD(16bit)的加法为例，在相同时钟周期内可以完成 4 次运算：



MMX 技术主要提供以下的特点：

- 八个 MMX 寄存器
- 三种新的数据类型
  1) 64bit packed byte data(signed,unsigned)
  2) 64bit packed word data(signed,unsigned)
  3) 64bit packed double word data(signed,unsigned)
- 支持新数据类型的新指令
  1) 数据传输指令（Data Transfer Instruction）

     MOVQ：可以将 64bit 的数据一次传输到 MMX 寄存器中，并在 MMX 寄存器之间进行数据传输。

     MOVD：可以将 32 位数据数据处传输到 MMX 寄存器的低 32bit 中，并在 MMX 寄存器之间进行数据传输。
  2) 算术运算指令（Arithmatic Instruction）
     a) PADDB，PADDW，PADDD 和 PSUBB，PSUBW，PSUBD

        对经过打包的 signed 和 unsigned 型的 Byte，Word，DoubleWord 整数进行 Wraround（存在溢出问题）加法计算。

        MOVQ mm0　0xFEFEFEFEFEFEFEFE;　//mm0(0) = 254

        PADDB mm0 2;　//mm0(0)　= 1
     b) PADDSB，PADDSW 和 PSUBSB，PSUBSW

        对经过打包的 signed 型的 Byte，Word 整数进行饱和（不存在溢出问题）加法运算。

MOVQ mm0　0xFEFEFEFEFEFEFEFE;　//mm0(0) = 0xFE

PADDB mm0 2;　//mm0(0)　= 0xFF

c) PADDUSB，PADDUSW 和 PSUBUSB，PSUBUSW

对经过打包的 unsigned 型的 Byte，Word 整数进行饱和（不存在溢出问题）加法运算。

d) PMULHW 和 PMULLW

打包 WORD 型数据的乘法运算。与加法运算不同，因为两个 WORD 相乘，结果是个 DWORD，则 MMX 寄存器明显不够用（4 个 WORD 变为 4 个 DOWRD），因此 Intel 采用了该方法：一次 WORD 乘法要进行两次计算，PMULHW 获得乘法的高位 WORD，PMULLW 获得乘法的低位 WORD。

对于图像处理中的 BYTE 乘法，MMX 没有给出相应的指令，目前的做法是：　　A*B

A7	A6	A5	A4	A3	A2	A1	A0
B7	B6	B5	B4	B3	B2	B1	B0

变换（1）为：

0	A7	0	A6	0	A5	0	A4
0	A3	0	A2	0	A1	0	A0

0	B7	0	B6	0	A5	0	A4
0	B3	0	A2	0	A1	0	A0

利用 PMULL 计算 A×B：

C7	C6	C5	C4
C3	C2	C1	C0

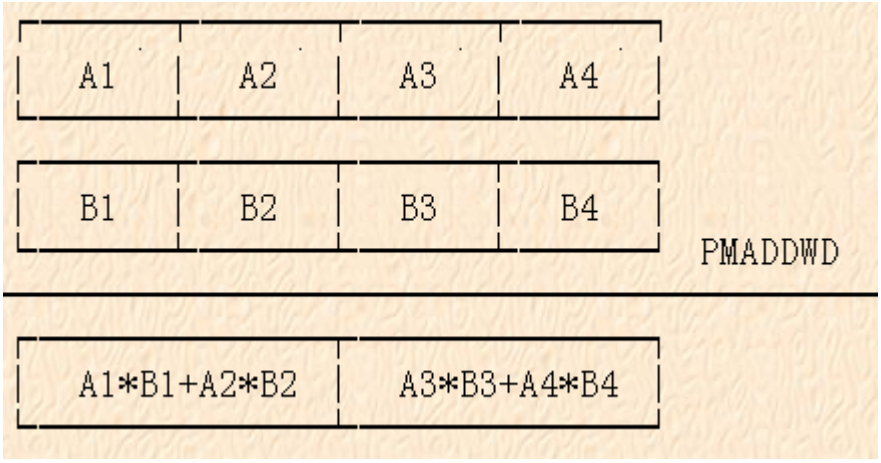
变换（2）：

C7	C6	C5	C4	C3	C2	C1	C0

由于 WORD 乘法运算的高 8 位都为 0，则计算得到的 DOWRD 的高 16 位也为 0，则直接通过 PMULL 就可以得到计算结果，而由于是 BYTE 的运算，则结果也必须是 BYTE，因此将 WORD 型结果饱和压缩为 BYTE 型。

e) PMADDWD

矢量运算



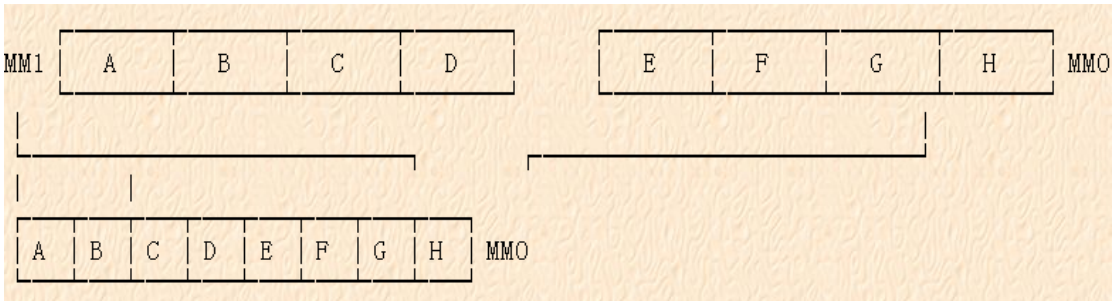
3) 打包解包指令（Pack Instruction & Unpack Instruction）

打包指令对数据存放格式进行各种整合，以用于进一步处理。

a) Packed 指令

PACKUSWB	有符号数 WORD 带饱和压缩成无符号 BYTE (pack word into bytes with unsigned saturation)
PACKSS[WB,DW]	有符号数带饱和压缩成有符号数 (pack word(dword) into bytes(word) with signed saturation)

PACKUSWB MM0, MM1:

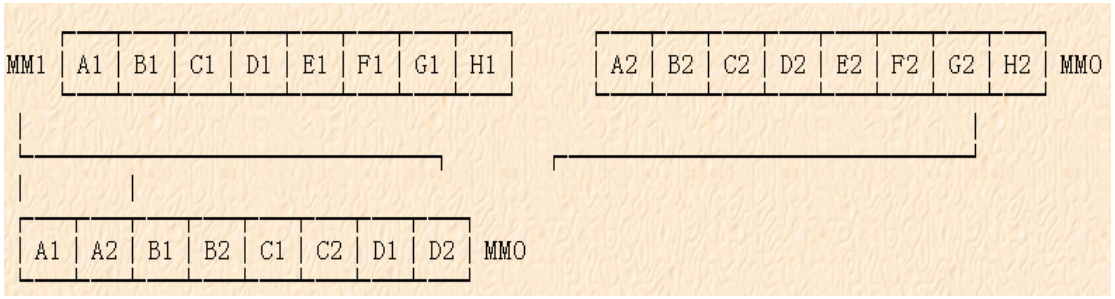


即上面的变换（2），在进行压缩（打包）时，饱和操作就是大于 255 的紧缩后就变成 255，小于 0 的紧缩后就变成 0。

b) Unpacked 指令

PUNPCKH[BW,WD,DQ]	交错放置两数的高位[byte->word,word->dword,dword->qword]
PUNPCKL[BW,WD,DQ]	交错放置两数的低位[byte->word,word->dword,dword->qword]

PUNPCKHBW MM0, MM1



即上面的变换（1），在变换 1 中 MM1 中为 0。

以下是将图像中每一个像素与一个系数相乘的例子：

```
movq    mm0,    [eax];    //eax 中存放图像数据低值
movq    mm1,    mm0;      //   mm0=mm1=A7A6A5A4A3A2A1A0
movq    mm2,    [ebx];    //eax 中存放乘法系数
movq    mm3,    mm2;    // mm2=mm3=B7B6B5B4B3B2B1B0

pxor    mm7,    mm7;   //mm7=0

punpcklbw     mm0,   mm7;    // mm0 = 0 A3 0 A2 0 A1 0 A0
punpckhbw     mm1,   mm7;     //mm1 = 0 A7 0 A6 0 A5 0 A4

punpcklbw     mm2,   mm7;    //mm2 = 0 B3 0 B2 0 B1 0 B0
punpckhbw     mm3,   mm7;    //mm3 = 0 B7 0 B6 0 B5 0 B4

pmullw  mm2,    mm0;         //C3 C2 C1 C0
pmullw  mm3,    mm1;          //C7 C6 C5 C4

packuswbmm2,    mm3;      //mm2=C7C6C5C4C3C2C1C0
```

4) EMMS 指令

由于 MMX 寄存器使用了浮点寄存器，因此在 MMX 代码中不能直接进行浮点数运算。如果要进行浮点运算或者 MMX 代码结束，必须使用 EMMS 指令对 x87FPU 进行置位。

5) 其他指令：

比较指令，

PCMPEQ[B,W,D]	串等于比较(byte,word,dword)
PCMPGT[B,W,D]	串大于比较(byte,word,dword)

逻辑指令，

PAND	按位与操作
PANDN	按位与后再取非操作
POR	按位或操作
PXOR	按位异或操作

移位指令

PSLL[W,D,Q]	逻辑左移[word,dword,qword]
PSRL[W,D,Q]	逻辑右移[word,dword,qword]
PSRA[W,D,Q]	算术右移[word,dword,qword]

SSE 指令集

SSE 在 PentiumIII 处理器中引入，主要针对 3D 处理中的浮点数运算，对 MMX 和 XMM 寄存器中的 Packed single-precision float 数进行操作，对 MMX 中的整数进行操作。并且 SSE 提供了状态控制，Cache 控制和内存排序等操作。

SSE 指令对 MMX 技术进行了扩展：

- 添加了 8 个 128bit 寄存器 XMM0－XMM7
- 一个 32bit 的 MXCSR 寄存器，用于为 XMM 中的运算提供控制位和状态。
- 128bit 单精度浮点数类型(将 IEEE 的单精度浮点数打包为一个 double quadword)
- 用于执行浮点数运算和整数运算的 SIMD 指令：
  - 128 位的 Packed 和 Scalar 单精度浮点数指令，用于进行 XMM 寄

存器运算。

Packed 和 Scalar 的区别：

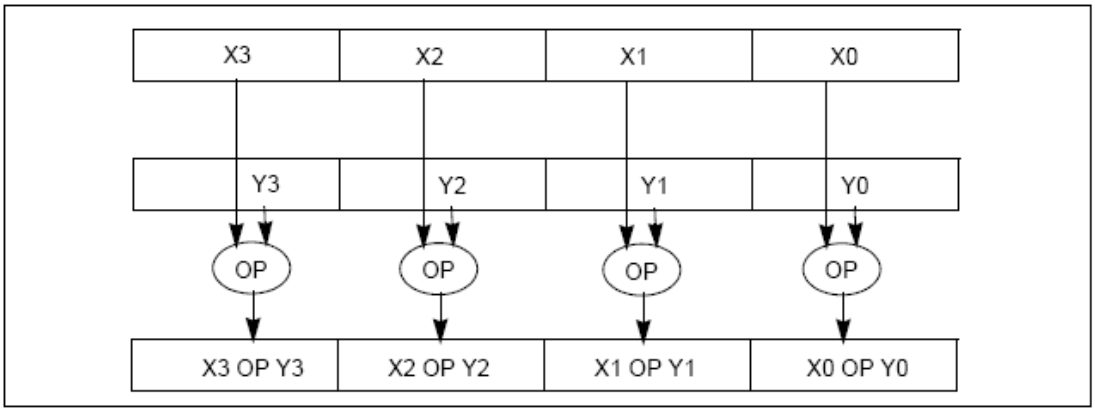


Figure 10-5. Packed Single-Precision Floating-Point Operation

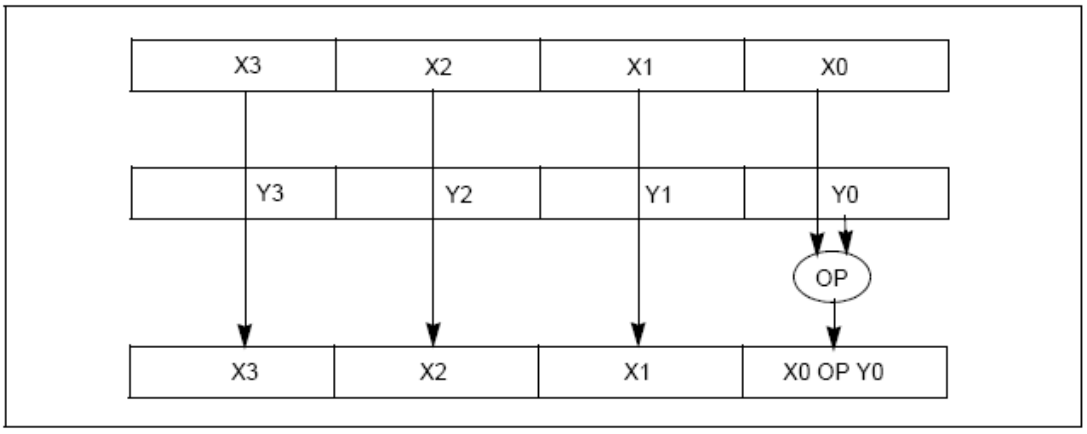


Figure 10-6. Scalar Single-Precision Floating-Point Operation

1) 数据传输指令（Data Transfer Instruction）

a) **MOVAPS**：move aligned packed single-precision float-point data

这里的 aligned 指如果源操作数在内存中，则该内存的首地址必须是 16 的倍数（在 MMX 指令中，必须为 4 的倍数），否则产生异常错误。在 Scalar 运算中没有此限制。

b) **MOVUPS**：move unaligned packed single-precision float-point data。

c) **…**

SSE 的数据传输指令提供了分别移动 XMM 高位部分和低位部分的指令。

2) 算术运算指令（Arithmatic Instruction）

提供了对于 Packed 和 Scalar 浮点数的加减乘除法、倒数、平方根、平方根倒数、最大最小值。

3) 变换指令

提供了将 packed DWORD 整数与 packed 单精度浮点数之间和 DWORD 整数与 scalar 单精度浮点数之间的变换。

4) SHUFFLE 和 Unpack 指令

SHUFFLE 实现了灵活的数据排列操作：

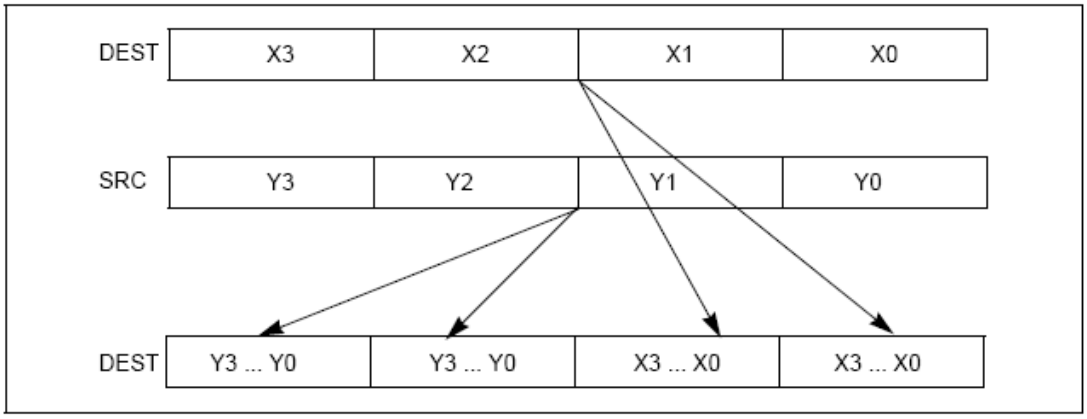


Figure 10-7. SHUFPS Instruction, Packed Shuffle Operation

UNPCKHPS 和 UNPCKLPS 与 MMX 指令类似。

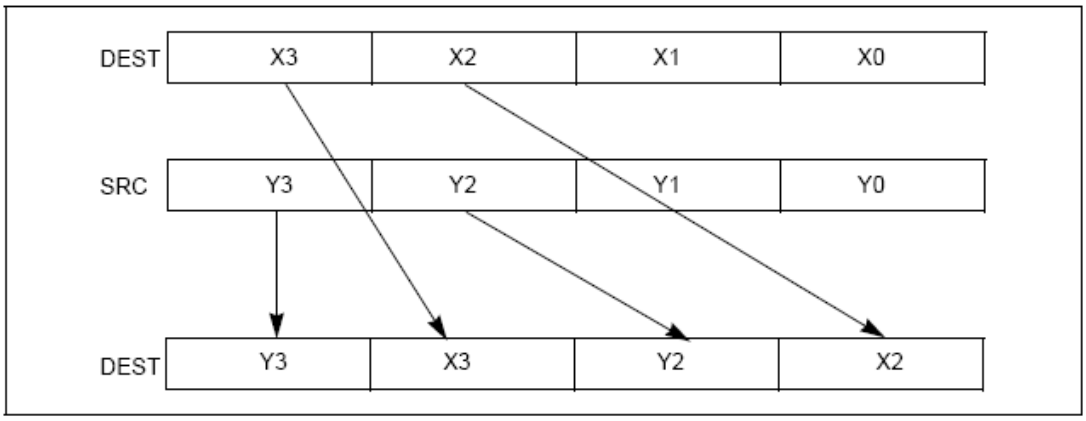


Figure 10-8. UNPCKHPS Instruction, High Unpack and Interleave Operation

◆ 64 位的 Packed 整数运算扩展指令。SSE 没有提出 128 位的整数指令，即 SSE 指令集不能用于计算 XMM 中的整数运算。

a) PEXTRW 将一个 WORD 从通用寄存器或者内存中传输到 MMX 中指定 WORD 位置。

b) PMOVMSKB，将 MMX 寄存器中每个 BYTE 的第 7 位转移到一个通用寄存器中低 8 位的每一位上。

PMOVMSKB *r32, mm*

PMOVMSKB instruction with 64-bit source operand:
r32[0] ← SRC[7];
r32[1] ← SRC[15];
* repeat operation for bytes 2 through 6;
r32[7] ← SRC[63];
r32[31-8] ← 000000H;

```
PXOR MM7, MM7                          // MM7 设置为零
MOVQ MM0, [SI]                         // 从内存中读数
PCMPEQB MM0, MM7            // 找零
PMOVMSKB EAX, MM0 // 将 8 位标志送入到通用寄存器中
```

该指令可以对变长编码运算（DCT 运算）进行优化，由于 MMX 的比较指令只在寄存器中产生结果，而不影响标志位，对于那些需要依赖运算结果而改变程序流程的算法就不太方便，通过使用 PMOVMSKB 指令，可以很方便将比较的结果传送到通用寄存器中。

c) PSADB 指令，计算各个数的绝对值的和

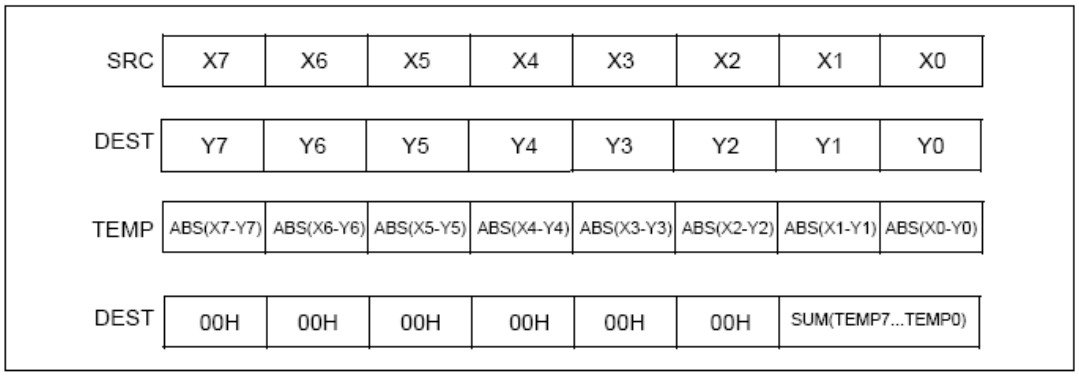


Figure 4-5. PSADBW Instruction Operation Using 64-bit Operands

d) 计算两个操作数中各个 Byte 或者 Word 平均值，最大最小值的指令。

e) PSHUFW 指令，可以实现源操作数中 WORD 顺序的任意排列：

```
DEST[15-0] ← (SRC >> (ORDER[1-0] * 16) )[15-0]
DEST[31-16] ← (SRC >> (ORDER[3-2] * 16) )[15-0]
DEST[47-32] ← (SRC >> (ORDER[5-4] * 16) )[15-0]
DEST[63-48] ← (SRC >> (ORDER[7-6] * 16) )[15-0]
```

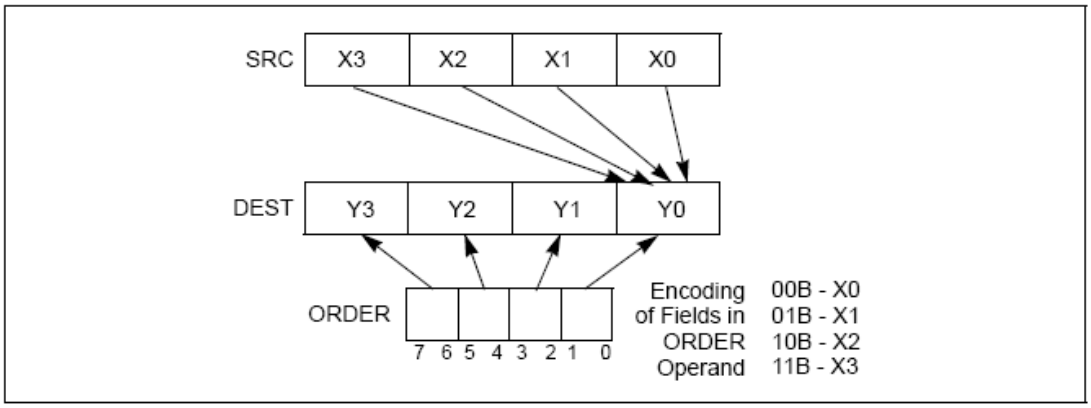


Figure 4-6. PSHUFD Instruction Operation

- 进行数据预读取（Prefetch），cache 控制和控制存储操作数顺序的指令。

  Prefetch 指令可以命令 CPU 提前将内存数据读取 Cache 中。

  MOVNTQ，MOVNTPS 指令命令 CPU 将计算结果直接写入内存中，而避免污染 Cache。

  MASKMOVQ：利用一个 Mask 将所指定的 BYTE 直接写入内存中，而避免污染 Cache。

  SFENCE 指令：

SSE2 指令集

SSE2 在 Pentium4 和 Xeon 处理器中引入，对 XMM 寄存器中的 Packed double-precision float 数进行操作，对 MMX 和 XMM 中的整数进行操作。SSE2 将在 MMX 中使用的整数指令扩展到 XMM 中，并提出了若干新的整数指令和新的 Cache 控制和内存排序指令。

SSE2 指令提供如下新特点：

- 新的数据类型

  4) 64bit packed double word data(signed,unsigned)128bit 的 packed 双精度浮点数（两个 IEEE 定义的双精度浮点数）

  5) 128bit BYTE，WORD，DWORD，QWORD 型整数

- 支持新数据类型的指令

1) Packed 和 Scalar 的双精度浮点数指令

将单精度浮点数指令用于双精度浮点数，并提供了两者的转换指令。

2) 附加的 64bit 和 128bit 整数指令

a) MOVDQA 和 MOVDQU：传输 aligned 和 unaligned 的 double word 整数

b) 128bit 的加减乘法指令，移位指令打包解包指令

c) 将数据在 MMX 和 XMM 中进行传输的指令。

d) PSHUFLW 和 PSHUFHW 将 SSE 中的 PSHUFW 指令分为高位操作和低位操作。

3) 将 MMX 和 SSE 中的整数指令扩展到 128bit。

所有的 MMX 和 SSE 中都扩展到 128bit 的 XMM 中进行操作，其指令基本相同。

4) 附加的 Cache 控制指令和指令顺序控制指令

a) CLFULSH：flush cache line

b) 将 SSE 中的 Cache 操作扩展到 128bit。

c) LFENCE 和 MFENCE 指令

SSE3 指令集

SSE3 在支持 Hyper-Thread 的 Pentium4 中引入，提高 SSE2，SSE 和 x87 的运算性能。

- 不对称运算指令和横向运算指令：

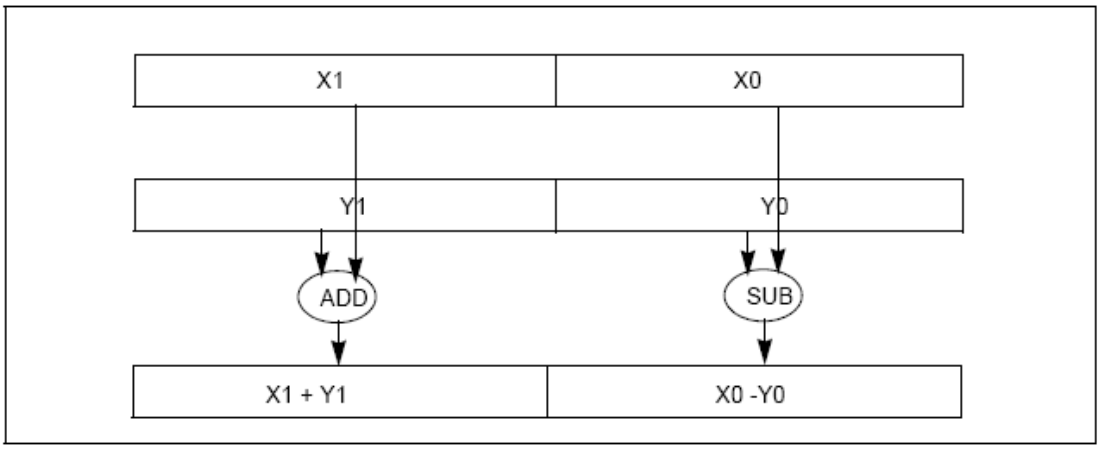


Figure 12-1. Asymmetric Processing in ADDSUBPD

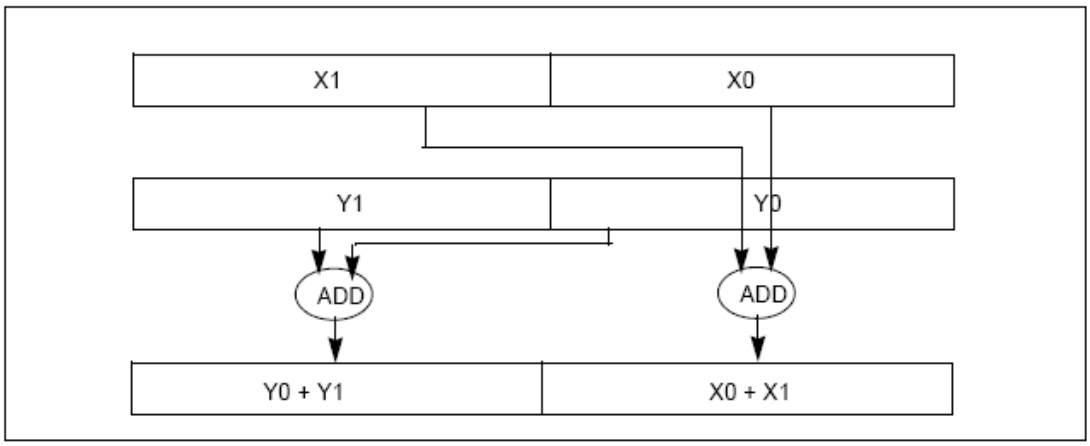


Figure 12-2. Horizontal Data Movement in ADDSUBPD

- 针对 128bit unaligned 数据载入的 SIMD 指令 LDDQU

  如果要读取的内存地址是 16 的整数倍，则直接读出 16 Byte，如果不是 16 的整数倍则读出 32 Byte，然后取出其中 16 位。该指令提高了读取内存的效率。

- 针对 128bit 浮点数的载入，存储，复制的优化指令。
- 线程控制指令

Extend memory 64 技术

增加了 8 个 XMM 寄存器 XMM8－XMM15，但是这 8 个寄存器必须在 64 位模式下采用使用。

三，目前 SIMD 的运行环境分析

目前主要开发程序对各种 SIMD 指令的支持

	VC6	VC7	Intel complier	MASM
MMX	×	×	×	×
SSE		×	×	×
SSE2		×	×	×
SSE3			×	

Instrincs－内联函数

内联函数是Intel为了能够使编程人员不用进行汇编编程就可以直接进行汇编语句调用的C函数。例如int abs(int)等。Intel将SIMD指令用Instrincs函数进行了一一对应的实现，例如：

```
__m64 x = _m_paddd(y, z);
float f = (_mm_empty(), init());
```

采用内联函数避免了繁琐的汇编编程，并且增强了程序的易读性。 以下是汇编与内联在读取内存中的处理。

MMX 汇编	SSE2 内联函数

mov eax, pImageMeshLine0;
add eax, i;
movq mm0, [eax];
movq mm1, mm0;
	n128Image_L =
	_mm_loadu_si128((__m128i*)(pImageMeshLine0 +
	i))

程序优化的各种工具

- Processor Pack，VC6 对 SIMD 的支持只有 MMX 汇编指令，因此，MS 针对 Intel 和 AMD 在 CPU 指令上的扩充，发布了 Processor Pack。在 VC6 中安装 Processor Pack，添加了 Intel 的内联函数支持并更新了 ML 编译器，可以直接支持 Intel SSE2 和 AMD 3DNow!™ 的汇编指令编程和内联函数编程。但是在编译中必须要对程序进行 MaxSpeed 级别的编译才能充分发挥其优越性。以下是对一个函数使用 SSE2 代码的测试时间：

编译类型	Debug	Release(default)	Release(MaxSpeed)
处理时间	50	50	17

同样的设定也适用于 VC7。

- Intel C++ Compiler，在 VC6 中可以使用 Intel C++ Compiler 替换 VC 中的 ML 编译器，达到使用最新的 SIMD 指令的目的。
- Intel IPP：对于固定的图像处理算法，视频处理算法，数字信号处理算法，矩阵处理算法等等，不需要花费时间进行编程，可以直接使用 Intel 提供的 Intel Integrated Performance Primitive 中的各种函数库直接进行处理。

Intel IPP 的三个函数库（视频音频，数学库，数字信号处理库）。以下是采用 Pentium M 处理器对 640*480 的图像采用 Sapara 4.1 和 IPP 做旋转（双线性插值）的处理时间

Sapara 4.1	IPP
9ms	2.3 ms

四，目前大张检查机项目中所使用的 SIMD 优化技术

大张检查机所面临的处理时间压力

大张检查机的图像数据为 1024×3264 的灰度图像，处理过程包括定位，与模板进行比较，掩模技术，Blob 图像分析技术。整个处理时间要求在 300ms 内完成，同时还必须保证系统处理时间有一定冗余度，因此处理时间非常紧张。

大张检查机目前所采用的算法

大张检查机主要针对具体算法，利用汇编指令采用 MMX 进行优化，主要在三个方面进行优化：

- 采用 MMX 指令进行并行计算。算法中的加法、减法和乘法都使用饱和 MMX 指令进行优化。但是单纯使用 MMX 代码进行模块化优化对效率的提高也是有限的，以下是几个处理函数采用 C 代码和 MMX 代码的比较:

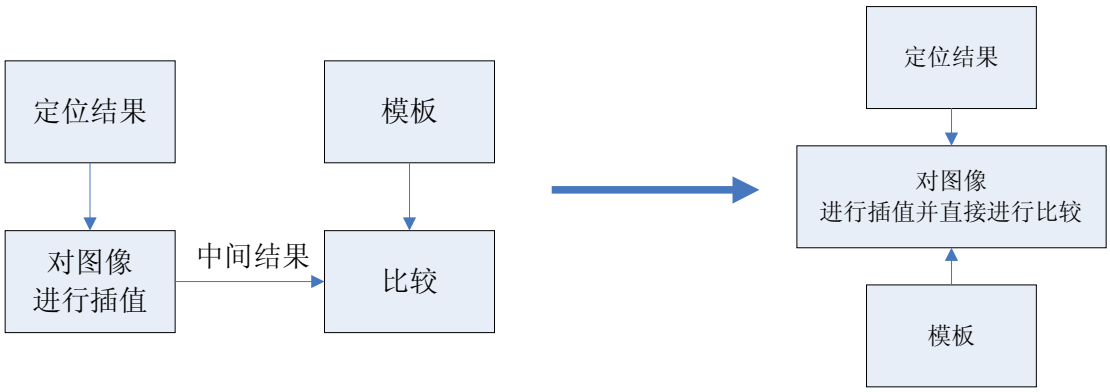
	C 代码	MMX 代码
模板打包（两个模板）	32ms	12ms
模板打包（四个模板）	60ms	28ms
白平衡代码	329ms	11ms
BuildFinalIModelsMMX	144ms	53ms

- 在使用 MMX 的同时，将所有的浮点数运算（插值、白平衡等操作）修改为整数运算，以避免浮点数运算的时间。从以上表中可以看到这种方法对于提高程序效率有极大的帮助。

  Int Fun(1.25) ->int (Fun(1) + Fun(0.25 * 255)/255)

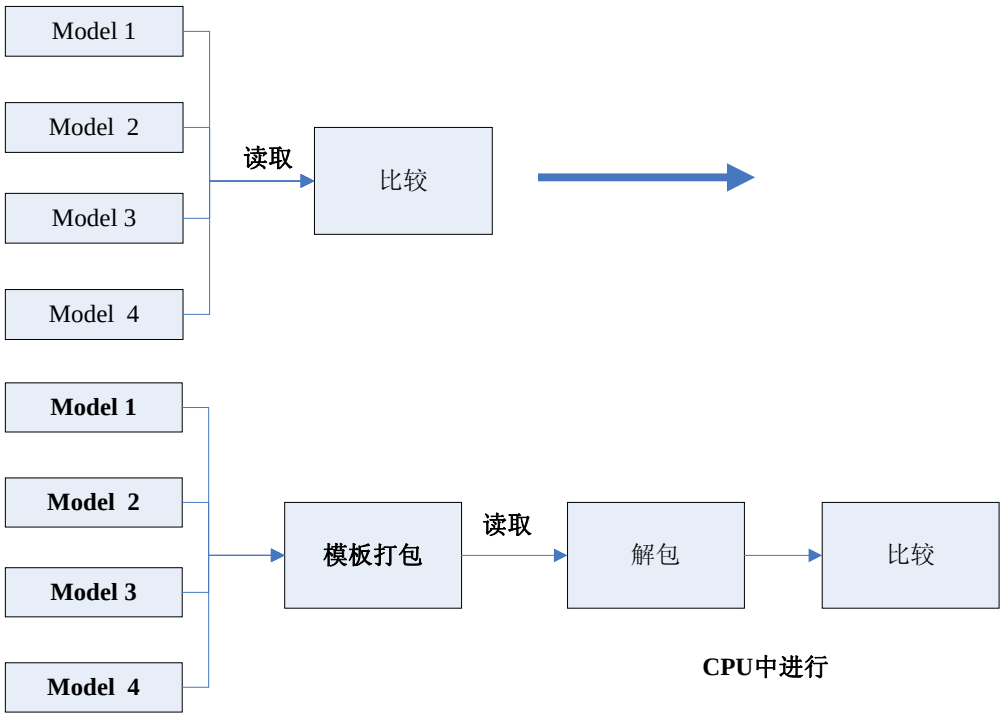
  其中 0.25 * 255 是预先计算得到的。整数的/255 操作在汇编中是一个>>8 的操作。

- 将处理流程中相邻并且关系密切的处理模块在寄存器级进行合并。例如：



采用这种合并方式，避免了中间结果的存储，节省了处理时间。

- 优化内存管理，减小内存读写次数。例如：



采用四个模板分别进行比较，需要读取四个不同的模板，在各个指针之间进行切换读取，会对 CPU 的 L2 Cache 进行反复操作，不利于流处理，而将模板进行打包后就可以只读取一个模板，实现了流处理，充分发挥了 L2 Cache 的缓存功能。

在研发阶段，采用各种优化方法对程序进行优化后，整个处理时间从 1.8s 左右下降到了 100ms 左右。

五，采用最新的 SIMD 技术对大张检查机项目进行优化

由于在研发时期，VC6 并不支持 SSE2 技术，并且 MMX 技术能够满足要求，因此没有考虑采用最新的 SIMD 技术对代码进行进一步优化。

但是随着大张检查机算法的不断升级，新的算法的加入又再次使处理时间达到了临界位置，因此有必要采用最新的 SIMD 技术进行二次优化。

由于 SSE3 推出的时间短，对其支持的软件少，因此目前采用 SSE2 指令对目前 MMX 代码进行二次优化，并进行了一些试验。

在图像处理中，SSE2 与 MMX 相比有几个新特点：可以进行 128 位的并行运算，可以对 Cache 进行控制，并且控制数据存储的状态。

采用 SSE2 指令进行的代码优化试验。

- 白平衡算法

```
for(int y = 0; y < Height;y++)
{
    for(int x = 0; x < Width; x++)
    {
        pImage(y * Width + x) =(float) pImage(y * Width + x) * pPara[x];
    }
}
```

测试环境：VC.net, PIV Willamette Processor, L2 cache 256KB 1.7GHz,L1 Cache data 8KB, code 12KB　P4B266C

指令类型	debug	Release(1)	Release(2)	Release(3)	Release(4)
MMX 汇编	26.3ms	25.3ms	25.1ms	25.5ms	25.3ms
MMX 内联函数	180ms	22ms	20.9ms	20.7ms	20.3ms
SSE2 指令，采用 XMM	17.9ms	17.9ms	18.5ms	18.3ms	17.7ms
SSE2 内联函数	112ms	16.7ms	16.7ms	16.7ms	16.4ms

(1), 其中 Release 的优化为自定义优化，处理器优化为“无列表”

(2), 其中 Release 的优化为最大化速度，处理器优化为“无列表”

(3) 其中 Release 的优化为最大化速度，处理器优化为“PIV”

(4), 其中 Release 的优化为自定义优化，处理器优化为“PIV”

该算法的特点是输入为 2 个，输出为 1 个，其中处理算法简单。优化方法为直接进行优化，未进行任何修改。新代码将老代码效率提高了 **35%**。

- 模板合并算法

```
for(int j = 0; j < m_nBufHeight*m_nBufWidth; j++)
{
    m_pOMixThrBufAddr[j * 4]     = m_pOHThrBufAddr[j];
    m_pOMixThrBufAddr[j * 4 + 1] = m_pOLThrBufAddr[j];
    m_pOMixThrBufAddr[j * 4 + 2] = m_pPHThrBufAddr[j];
    m_pOMixThrBufAddr[j * 4 + 3] = m_pPLThrBufAddr[j];
}
```

测试环境：VC 6＋Processor Pack，迅驰处理器，L2 2M。＝

	OMix
老代码（MMX） ：	28.3ms
新代码（SSE2 内联函数）1 ：	27ms
新代码（SSE2 内联函数）2 ：	17.3ms

该算法的特点是输入为 4 个指针，输出为一个指针，中间有若干打包拆包过程。优化方法 1 将老代码中的 MMX 代码直接更换为 SSE2 代码，发现处理时间几乎不变，然后使用了优化方法 2：用 MOVNTQ 取代 MOVDQA，将处理结果直接写入 Cache，其他不变。新代码应用于 2 个输入指针和四个输入指针时，效率分别提高了 **24%**和 **38%**。

- 插值相减算法

	JOMixThrMMX
老代码（MMX） ：	28.3ms
新代码（SSE2 内联函数） ：	27.8ms

该算法的特点是在函数开始输入一个指针，其后输入另一个指针，最后输出四个指针。采用以上的两种方法都没有明显的效果，需要进一步进行分析和探讨。

优化中的心得

- SSE2 指令对 MMX 指令的二次优化不会是 128/64＝2 的结果，在很多复杂算法下，直接将 SSE2 代码替换 MMX 指令并没有效果。

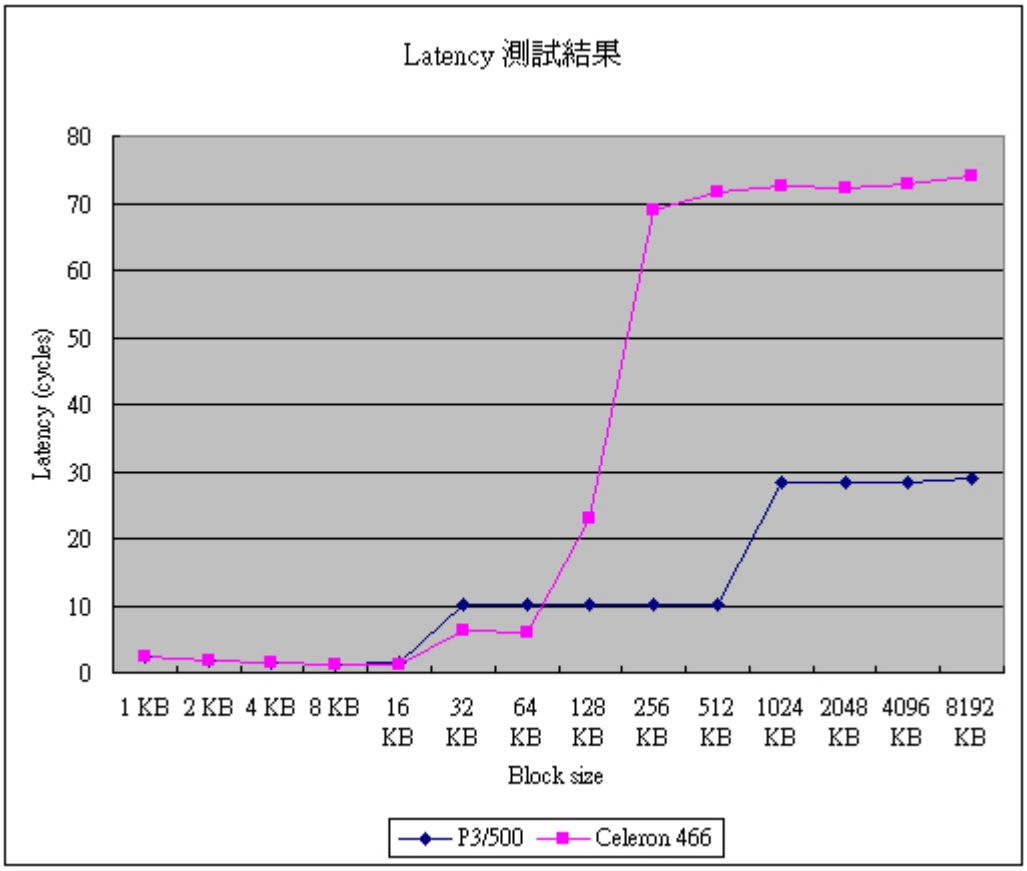
- 目前运算时间最重要的瓶颈不是加减乘除法的次数，而是内存和 CPU 之间的数据传输的速度。以下是针对中间处理过程和内存读写的测试结果：

	JOMix
完整处理：	27.8ms
去掉中间所有的过程（50％语句），仅仅保留内存读写语句：	27ms

从试验中可以看到，内存的读写在处理时间中占有绝大部分。

- 缓存是加快代码速度的重要途径

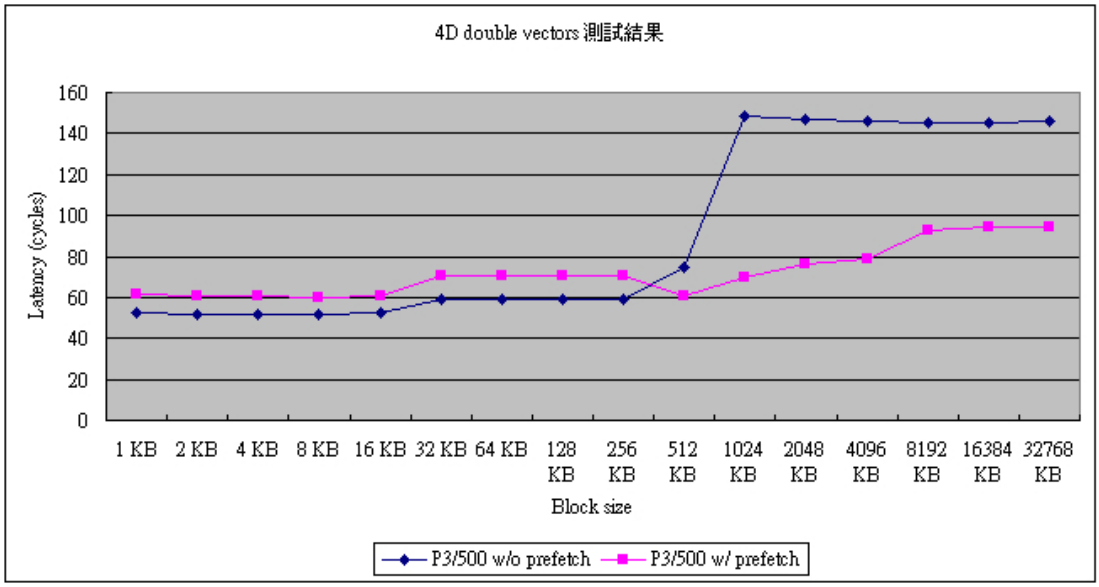
由于内存与 CPU 之间的传输速度很慢（CPU 与 Cache 之间的交换速度＝CPU 频率，CPU 与内存的交换速度＝CPU 频率/倍频）。以下是 Hotball 编写的一个连续读取内存块的程序在大 Cache PIII 和小 Cache Celeron 中的表现曲线。



在曲线上可以看出，在大数据量处理中，如何让 CPU 尽可能少的直接读取内存，而直接在 CacheL1 和 L2 中读取数据，是代码优化的一个关键点。

以下是 Hotball 编写针对 4D 运算的有无 Cache 控制的程序结果，采用了

Prefetch 对程序进行预读取。



在对大张检查机进行的试验中，由于使用的 Buffer 很多，采用将结果直接写入内存避免污染 Cache 的方法，对程序的执行效率有着明显的提高。

总的来说，采用大容量 L2 Cache 的 CPU 对程序的处理速度有着明显的改善。以下是对大张检查机程序中的某个处理函数在现场工控机上和笔记本上的运行时间对比，可以看到优化指令，Cache 和前端总线频率对程序效率的主要程度远远大于了 CPU 的频率。

2.8G PIV CPU， 512K L2 Cache 采用 MMX 技术，266M 前端总线	34ms
1.5G P M CPU，2M L2 Cache 采用 MMX 技术，400M 前端总线	20ms
1.5G P M CPU，2M L2 Cache 采用 SSE2 技术，400M 前端总线	17ms