Vectorization

Vectorization diagram

```
for(i=0; i < n; i++){
   b[i]=a[i]*c[i]
}
```

Step 1: b[0]=a[0]*c[0]
Step 2: b[1]=a[1]*c[1]
Step 3: b[2]=a[2]*c[2]
Step 4: b[3]=a[3]*c[3]
Step 5: b[4]=a[4]*c[4]
Step 6: b[5]=a[5]*c[5]
Step 7: b[6]=a[6]*c[6]
Step 8: b[7]=a[7]*c[7]
....

```
for(i=0; i < n; i++){
   b[i]=a[i]*c[i]
}
```

Step 1: b[0:3]=a[0:3]*c[0:3]
Step 2: b[4:7]=a[4:7]*c[4:7]
,,,,

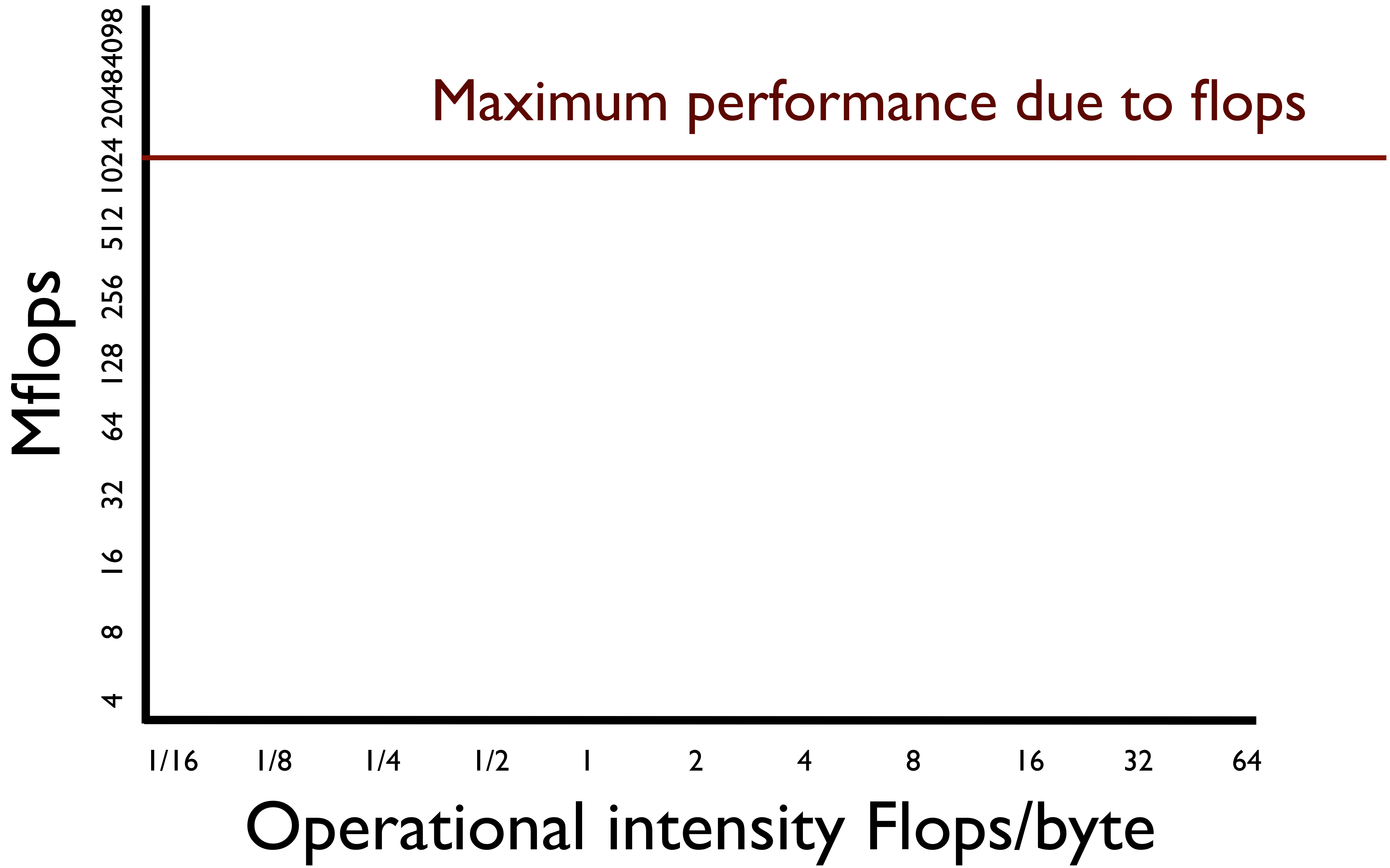
History of vector computers

- First research done in the 1960s at Westinghouse
- First commercial version was the Cray-1 in mid-70s
- Dominant supercomputing platform in late 70s to early 90s (Cray & Convex dominant players)

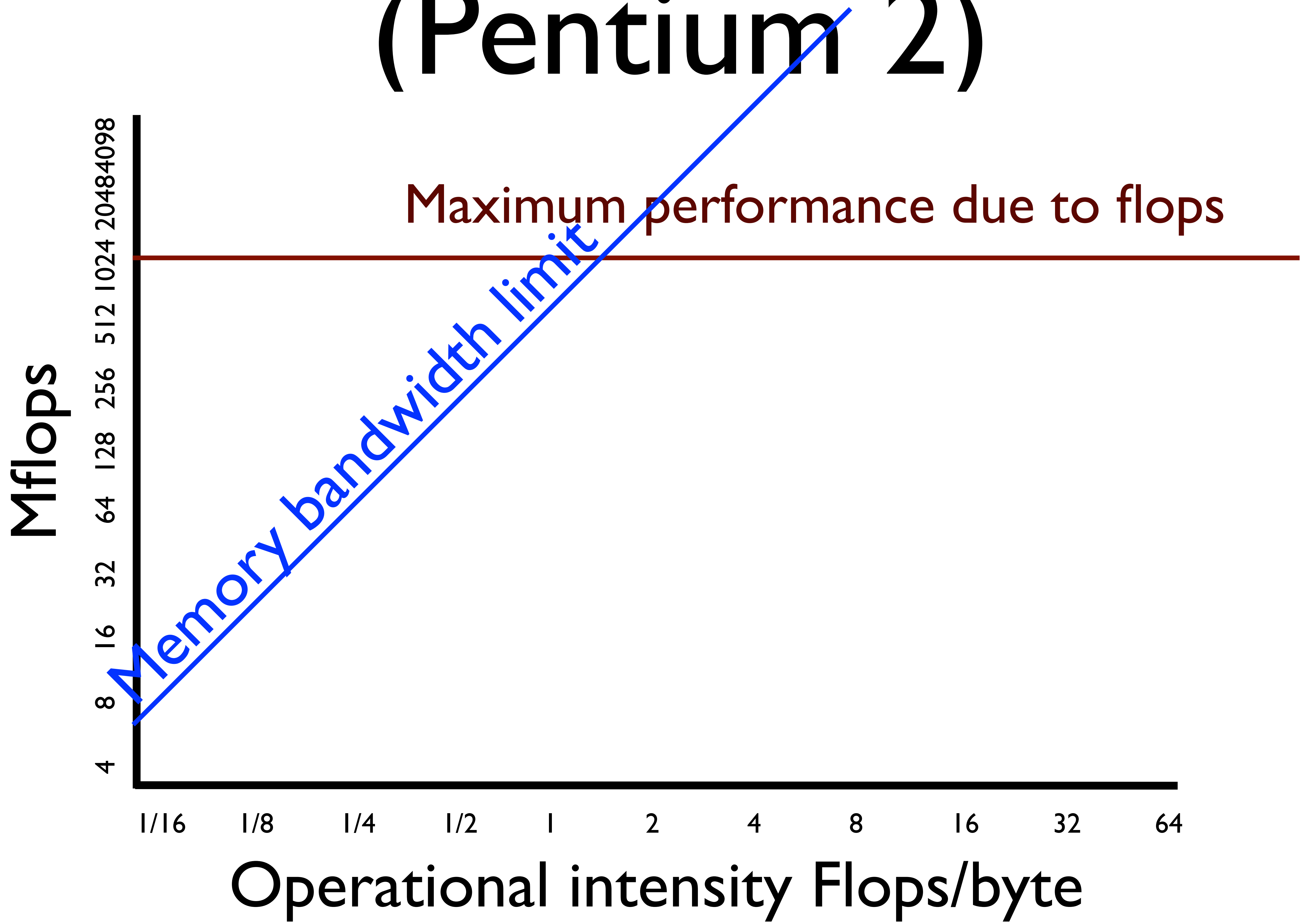
History of vector computers(2)

- Virtually disappeared in mid-90s with the advent of cheap commodity hardware
- Reintroduced into commodity hardware with the Pentium3
- Since then vector length has increased (8 in SandyBridge, 16 in XeonPhi)

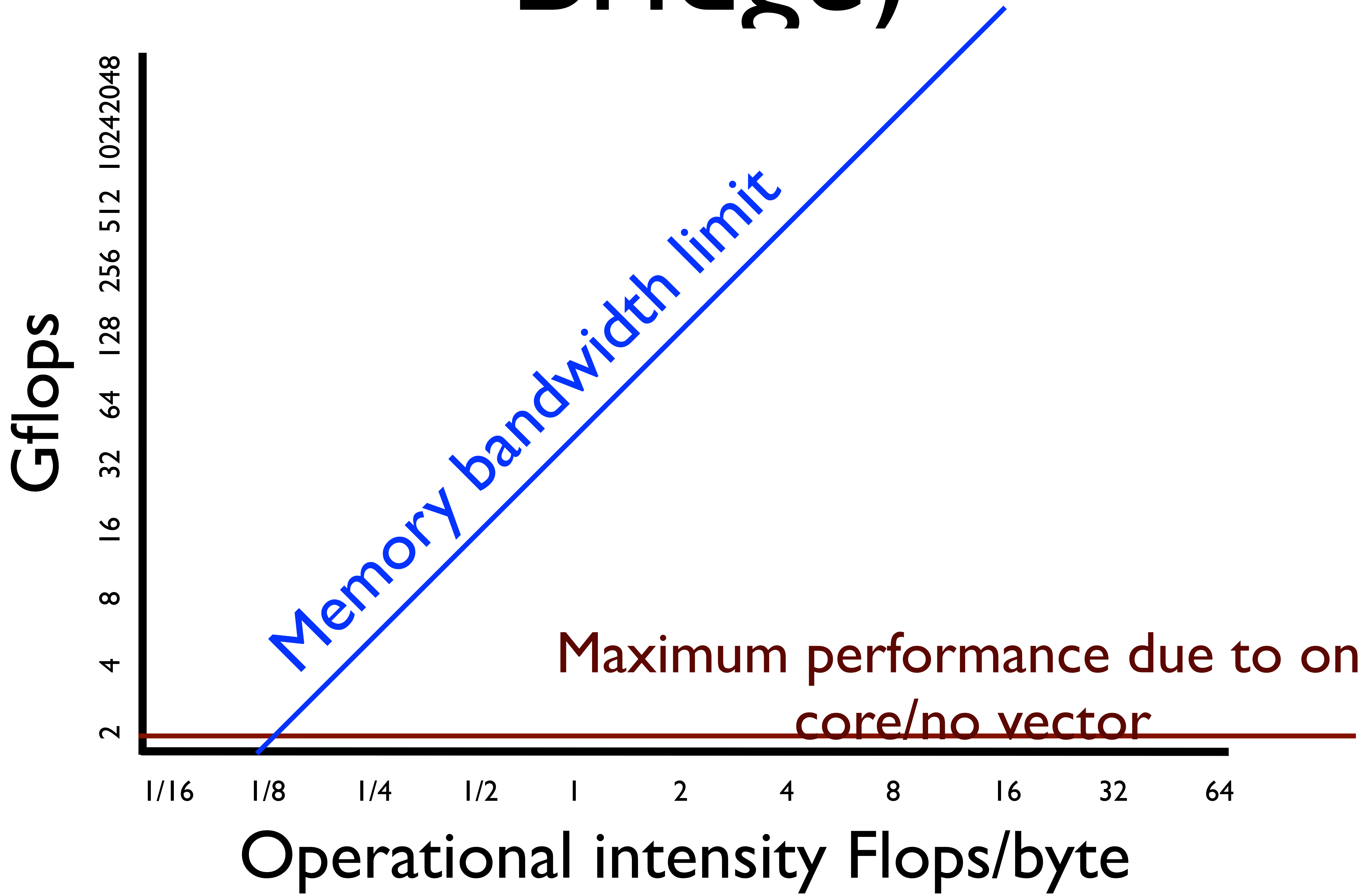
Roofline model (Pentium 2)



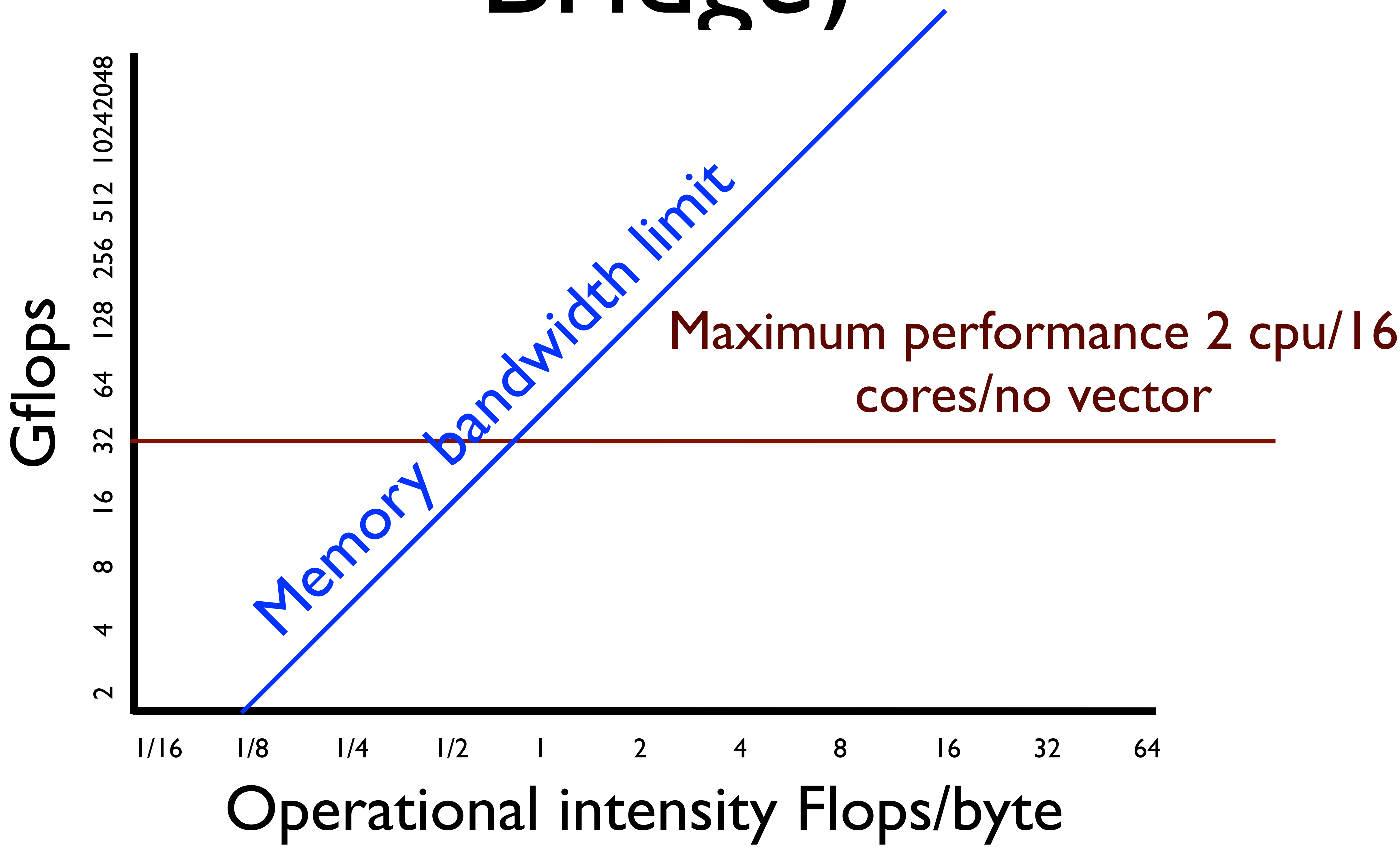
Roofline model (Pentium 2)



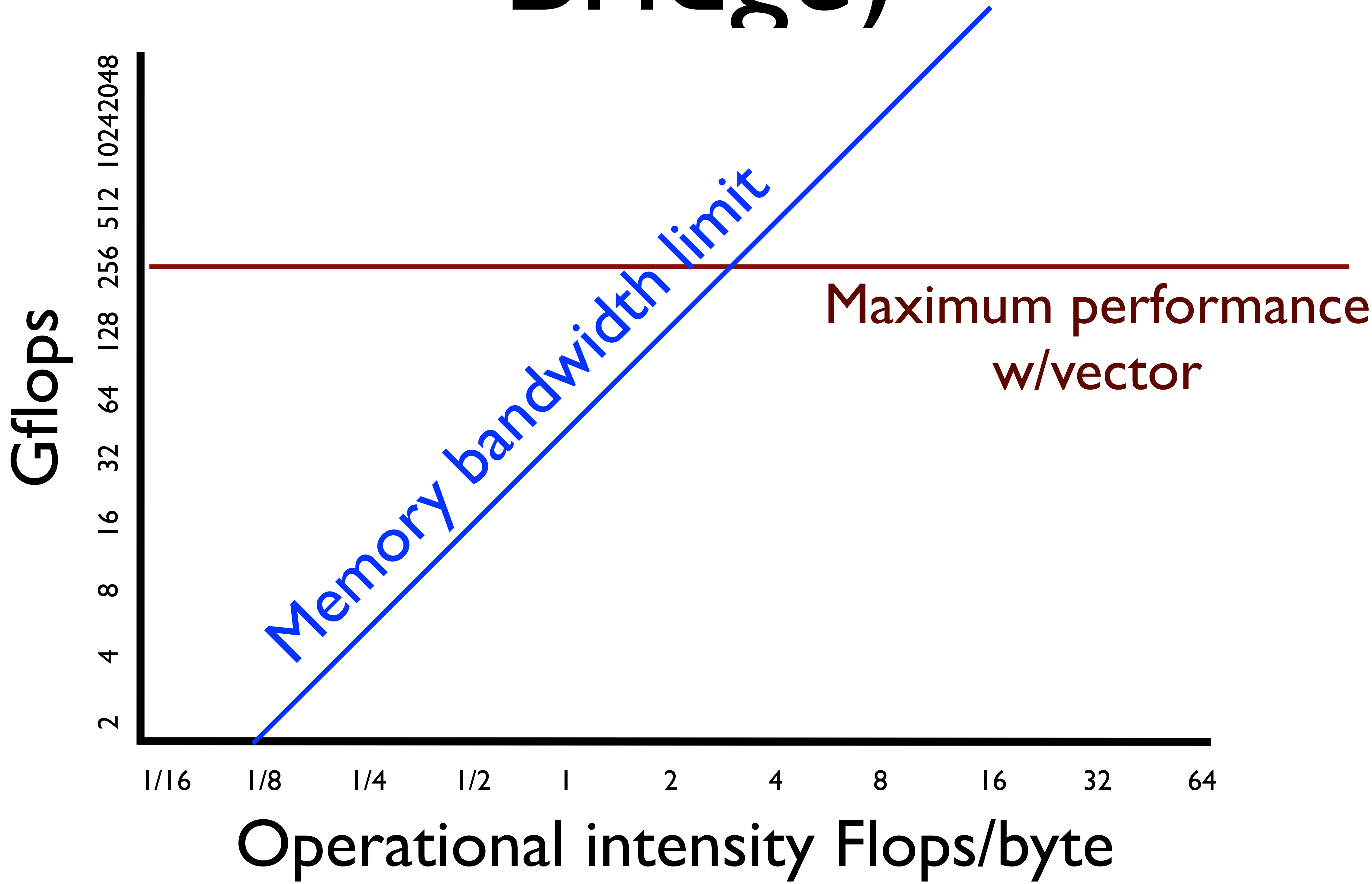
Roofline model (Sandy Bridge)



Roofline model (Sandy Bridge)



Roofline model (Sandy Bridge)



Assembler example

```
void mul_asm(float* out, float* in, unsigned int leng)
{    unsigned int count, rest;

     //compute if array is big enough for vector operation
     rest  = (leng*4)%16;
     count = (leng*4)-rest;

    // vectorized part; 4 floats per loop iteration
     if (count>0){
     __asm __volatile__  (".intel_syntax noprefix\n\t"
     "loop:                 \n\t"
     "movups xmm0,[ebx+ecx] ;loads 4 floats in first register (xmm0)\n\t"
     "movups xmm1,[eax+ecx] ;loads 4 floats in second register (xmm1)\n\t"
     "mulps xmm0,xmm1       ;multiplies both vector registers\n\t"
     "movups [eax+ecx],xmm0 ;write back the result to memory\n\t"
     "sub ecx,16            ;increase address pointer by 4 floats\n\t"
     "jnz loop              \n\t"
     ".att_syntax prefix    \n\t"
       : : "a" (out), "b" (in), "c"(count), "d"(rest): "xmm0","xmm1");
     }

     // scalar part; 1 float per loop iteration
     if (rest!=0)
     {
      __asm __volatile__  (".intel_syntax noprefix\n\t"
     "add eax,ecx           \n\t"
     "add ebx,ecx           \n\t"

     "rest:                 \n\t"
     "movss xmm0,[ebx+edx]  ;load 1 float in first register (xmm0)\n\t"
     "movss xmm1,[eax+edx]  ;load 1 float in second register (xmm1)\n\t"
     "mulss xmm0,xmm1       ;multiplies both scalar parts of registers\n\t"
     "movss [eax+edx],xmm0  ;write back the result\n\t"
     "sub edx,4             \n\t"
     "jnz rest              \n\t"
     ".att_syntax prefix    \n\t"
       : : "a" (out), "b" (in), "c"(count), "d"(rest): "xmm0","xmm1");
```

Vectorization code: Data independence

```
for(i=0;i<N;i++){
    a[i+1] = a[i]*b[i];
}
```

Cyclic dependence

```
for(i=0;i<N;i++){
    d[i] = a[i-1]*d[i];
    a[i] = b[i]+c[i];
}
```

Backward dependence

Loop distribution

```
for(i=0; i < n; i++){
  S1
  S2
  S3
}
```

```
for(i=0; i < n; i++){
  S1
}
for(i=0; i < n; i++){
  S2
}
for(i=0; i < n; i++){
  S3
}
```

Backward dependance

```
for(i=1;i< n; i++){
   d[i]=a[i-1]*d[i];
   a[i]=b[i]*c[i];
}
```

Backward dependance

```
for(i=1;i< n; i++){
   d[i]=a[i-1]*d[i];
   a[i]=b[i]*c[i];
}
```

=

```
for(i=1;i< n; i++){
   d[i]=a[i-1]*d[i];
}
   for(i=1;i< n; i++){
     a[i]=b[i]*c[i];
}
```

Backward dependance

```
for(i=1;i< n; i++){
   d[i]=a[i-1]*d[i];
   a[i]=b[i]*c[i];
}
```

```
for(i=1;i< n; i++){
   d[i]=a[i-1]*d[i];
}
   for(i=1;i< n; i++){
     a[i]=b[i]*c[i];
}
```

```
for(i=1;i< n; i++){
   a[i]=b[i]*c[i];
  d[i]=a[i-1]*d[i];
}
```

Loop sectioning

- The length of the registers often don't match the number of loop iterations
- Loops are broken (sectioned) into several loops (the length corresponding to the size of the vector registers) and a remainder loop
- The shorter the remainder loop the better the vectorization performance

Vector length

- Each operation in the vector ALU takes the same amount of time regardless of how full the vector registers are
- Filling more of the vector leads to better performance
- Performance tools can report the average vector length

How do you vectorize?

- Code design
- Pragmas
- ISPC
- Cilk+
- Guided vectorization(not covered)
- Intel MKL
- SSE/AVX

Ease

Flexibility/ performance?

Definitely beyond the scope of this class

Code design

- To get automatic vectorization
  - use at least -O2
  - make loops simple
  - check what is vectorized using -vec_report

-vec-reportn

- Work in C, C++, or Fortran
- Higher values of n give more information
- Will report which loops have been vectorized and which have not

Vec report example

```
subroutine quad(len,a,b,c,x1,x2)
  real(4) a(len),b(len), c(len), x1(len), x2(len), s

  do i=1,len
    s = b(i)**2 - 4.*a(i)*c(i)
    if (s.ge.0.) then
      x1(i) = sqrt(s)
      x2(i) = (-x1(i) - b(i)) *0.5 / a(i)
      x1(i) = ( x1(i) - b(i)) *0.5 / a(i)
    else
      x2(i)=0.
      x1(i)=0.
    endif
  enddo
end
```

```
> ifort -c -vec-report2 quad.f90
quad.f90(4): (col. 3) remark: LOOP WAS VECTORIZED.
```

Vec report example

```
subroutine no_vec(a, b, c)
  real(4), dimension(*) :: a, b, c
  integer :: i

  do i=1,100
     a(i) = b(i) * c(i)
     if (a(i) < 0.0 )  exit
  enddo

end

> ifort -c -vec-report2 two_exits.f90
two_exits.f90(5): (col. 3) remark: loop was not
vectorized: nonstandard loop is not a vectorization
candidate.
```

Pragmas

- Pragma give the compiler guidance (not guaranteed to be listened to)
- Fortran

 !$DIR option

- C

#pragma option

Pragmas

- loop count (n) : typical size of the loop (help decide whether to parallelize)
- vector always (safe to vectorize)
- vector align (assert data is aligned)
- novector (don't vectorize)
- vector nontemporal (data will not be reused)
- ivdep (ignore some vectorization killers)

Pragmas: simd

```
[D:/simd] cat example1.f
subroutine add(A, N, X)
integer N, X
real       A(N)
!DIR$ SIMD
DO I=X+1, N
    A(I) = A(I) + A(I-X)
ENDDO
end

Command line entry:  [D:\simd] ifort example1.f
-nologo -Qvec-report2
Output:  D:\simd\example1.f(7): (col. 9) remark:
LOOP WAS VECTORIZED.
```

Pragmas: simd

```
[D:/simd] cat example1.f
subroutine add(A, N, X)
integer N, X
real       A(N)
DO I=X+1, N
    A(I) = A(I) + A(I-X)
ENDDO
end
```

```
Command line entry: [D:/simd] ifort example1.f -
nologo -Qvec-report2
```

```
Output:  D:\simd\example1.f(6): (col. 9) remark:
loop was not vectorized: existence of vector
dependence.
```

Cilk

- Yet another parallel language developed at MIT
- Basic idea is to expose parallelism in the program
- Keywords
  - spawn - procedure call it modifies can safely operate in parallel
  - sync - cannot proceed until previous spawn are completed

Cilk example

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
    }
}
```

Cilk++

- Extension designed by intel to expose vectorization to the compiler
- Only in C,C++

cilk++

```
for(j=0; j < 8; j++){
        tmp1[j*8:8]=pcur[ind+ind1[:]]*sc1[:];
        tmp2[j*8:8]=pcur[ind+ind2[:]]*sc2[:];
        tmp3[j*8:8]=pcur[ind+ind3[:]]*sc3[:];
        ind++;
 }
 tmp4[:]=0.;
 tmp4[:]=tmp1[0:8:8]+tmp2[0:8:8]+tmp3[0:8:8];
 tmp4[:]=tmp1[1:8:8]+tmp2[1:8:8]+tmp3[1:8:8];
```

Intel Math Kernel Library

- Highly optimized library for mathematical functions
- VML (vector math library)
- Reference: http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-59EC4B87-29C8-4FB4-B57C-D269E6364954.htm

Intel Math kernel library

- Basically you redesign your code to a series of vector calls

```
do i=1,n
  b[i]=cos(c[i])
  a[i]=b[i]*c[i]+d[i]
end do
```

```
call vscos(n,c,b)
call vsmul(n,b,c,m)
call vsadd(n,d,m,a0
```

SSE/AVX

- Hundreds of function calls that allow the programmer to interact with a processor vector units
- SSE - Pre-SandyBridge
- AVX- +SandyBridge
- Reference:http://software.intel.com/sites/default/files/m/3/f/c/1/9/21558-

Writing AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Needed include file

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Needed include file

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Aligned_malloc
make sure memory is allocated at the most efficient memory boundary (such as beginning of a cache line)

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Special vector object (in this case 4 bytes together)

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Set all values to 4

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Break loop into n/4

Writing SSE/AVX

```
/*
     SSE_Tutorial
     This tutorial was written for supercomputingblog.com
     This tutorial may be freely redistributed provided this header remains
intact
*/

#include "stdafx.h"
#include <xmmintrin.h>   // Need this for SSE compiler intrinsics
#include <math.h>        // Needed for sqrt in CPU-only version


int main(int argc, char* argv[])
{
     printf("Starting calculation...\n");

     const int length = 64000;

     // We will be calculating Y = Sin(x) / x, for x = 1->64000

     // If you do not properly align your data for SSE instructions, you
may take a huge performance hit.

     float *pResult = (float*) _aligned_malloc(length * sizeof(float), 16);
// align to 16-byte for SSE

     __m128 x;
     __m128 xDelta = _mm_set1_ps(4.0f);      // Set the xDelta to (4,4,4,4)
     __m128 *pResultSSE = (__m128*) pResult;


     const int SSELength = length / 4;

     for (int stress = 0; stress < 100000; stress++)   // lots of stress
loops so we can easily use a stopwatch

     {
#define TIME_SSE    // Define this if you want to run with SSE
#ifdef TIME_SSE
          x = _mm_set_ps(4.0f, 3.0f, 2.0f, 1.0f); // Set the initial values
of x to (4,3,2,1)
```

Assign vector object (note reverse order)

```
    for (int i=0; i < SSELength; i++)
        {

            __m128 xSqrt = _mm_sqrt_ps(x);
            // Note! Division is slow. It's actually faster to take the reciprocal of a number and multiply
            // Also note that Division is more accurate than taking the reciprocal and multiplying

#define USE_DIVISION_METHOD
#ifdef USE_FAST_METHOD
            __m128 xRecip = _mm_rcp_ps(x);

            pResultSSE[i] = _mm_mul_ps(xRecip, xSqrt);
#endif //USE_FAST_METHOD
#ifdef USE_DIVISION_METHOD
            pResultSSE[i] = _mm_div_ps(xSqrt, x);

#endif  // USE_DIVISION_METHOD

            // NOTE! Sometimes, the order in which things are done in SSE may seem reversed.
            // When the command above executes, the four floating elements are actually flipped around
            // We have already compensated for that flipping by setting the initial x vector to (4,3,2,1) instead of
(1,2,3,4)

            x = _mm_add_ps(x, xDelta);    // Advance x to the next set of numbers

        }
#endif  // TIME_SSE
#ifndef TIME_SSE
        float xFloat = 1.0f;
        for (int i=0 ; i < length; i++)
        {

            pResult[i] = sqrt(xFloat) / xFloat;    // Even though division is slow, there are no intrinsic functions
like there are in SSE
            xFloat += 1.0f;
        }

#endif  // !TIME_SSE
```

```
    for (int i=0; i < SSELength; i++)
        {

            __m128 xSqrt = _mm_sqrt_ps(x);
            // Note! Division is slow. It's actually faster to take the reciprocal of a number and multiply
            // Also note that Division is more accurate than taking the reciprocal and multiplying

#define USE_DIVISION_METHOD
#ifdef USE_FAST_METHOD
            __m128 xRecip = _mm_rcp_ps(x);

            pResultSSE[i] = _mm_mul_ps(xRecip, xSqrt);
#endif //USE_FAST_METHOD
#ifdef USE_DIVISION_METHOD
            pResultSSE[i] = _mm_div_ps(xSqrt, x);

#endif  // USE_DIVISION_METHOD

            // NOTE! Sometimes, the order in which things are done in SSE may seem reversed.
            // When the command above executes, the four floating elements are actually flipped around
            // We have already compensated for that flipping by setting the initial x vector to (4,3,2,1) instead of
(1,2,3,4)

            x = _mm_add_ps(x, xDelta);    // Advance x to the next set of numbers

        }
#endif  // TIME_SSE
#ifndef TIME_SSE
        float xFloat = 1.0f;
        for (int i=0 ; i < length; i++)
        {

            pResult[i] = sqrt(xFloat) / xFloat;    // Even though division is slow, there are no intrinsic functions
like there are in SSE
            xFloat += 1.0f;
        }

#endif  // !TIME_SSE
```

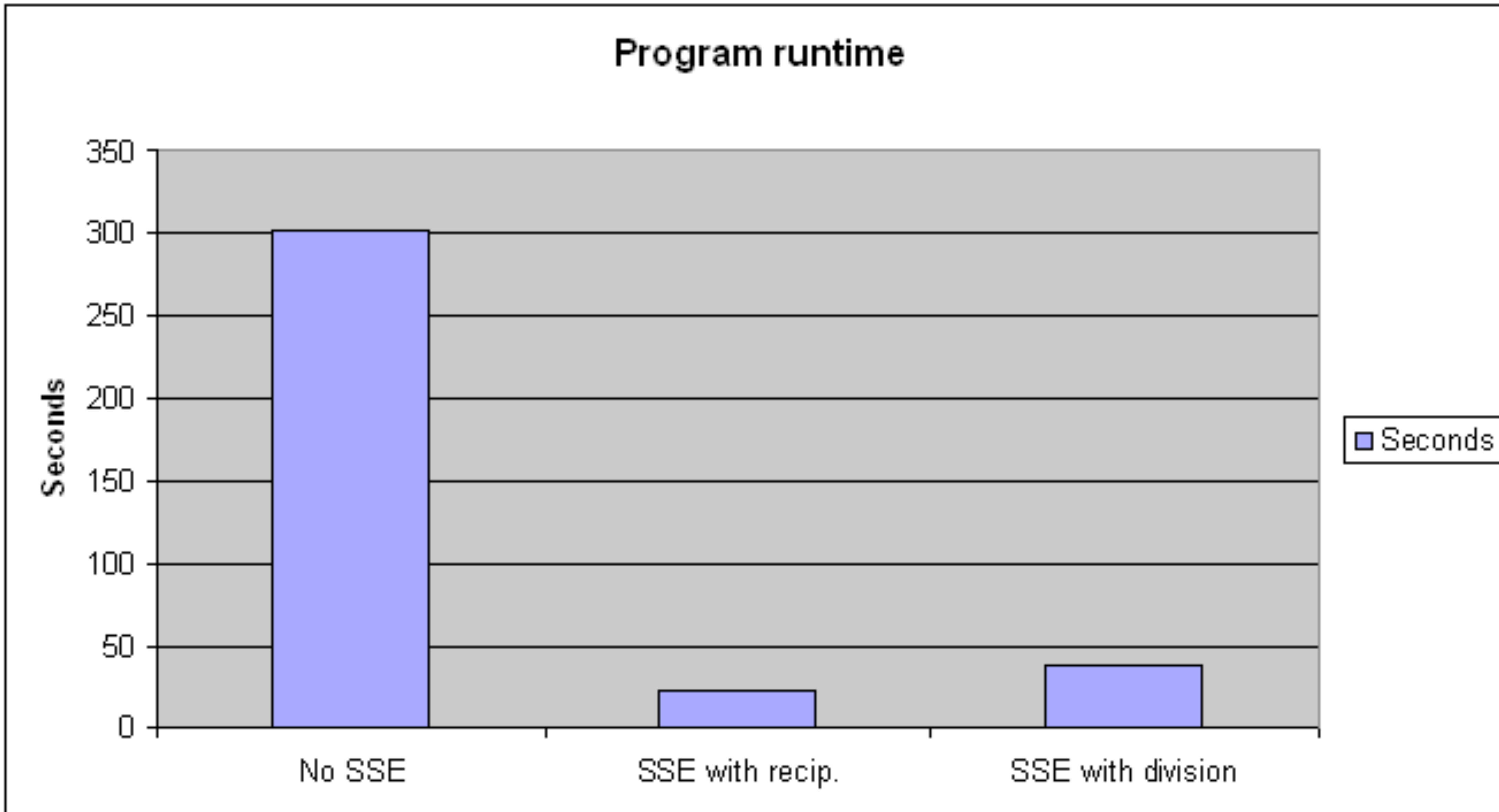
```
    for (int i=0; i < SSELength; i++)
        {

            __m128 xSqrt = _mm_sqrt_ps(x);
            // Note! Division is slow. It's actually faster to take the reciprocal of a number and multiply
            // Also note that Division is more accurate than taking the reciprocal and multiplying

#define USE_DIVISION_METHOD
#ifdef USE_FAST_METHOD
            __m128 xRecip = _mm_rcp_ps(x);

            pResultSSE[i] = _mm_mul_ps(xRecip, xSqrt);
#endif //USE_FAST_METHOD
#ifdef USE_DIVISION_METHOD
            pResultSSE[i] = _mm_div_ps(xSqrt, x);

#endif  // USE_DIVISION_METHOD

            // NOTE! Sometimes, the order in which things are done in SSE may seem reversed.
            // When the command above executes, the four floating elements are actually flipped around
            // We have already compensated for that flipping by setting the initial x vector to (4,3,2,1) instead o
(1,2,3,4)

            x = _mm_add_ps(x, xDelta);    // Advance x to the next set of numbers

        }
#endif  // TIME_SSE
#ifndef TIME_SSE
        float xFloat = 1.0f;
        for (int i=0 ; i < length; i++)
        {

            pResult[i] = sqrt(xFloat) / xFloat;    // Even though division is slow, there are no intrinsic functions
like there are in SSE
            xFloat += 1.0f;
        }

#endif  // !TIME_SSE
```

```
    for (int i=0; i < SSELength; i++)
        {

            __m128 xSqrt = _mm_sqrt_ps(x);
            // Note! Division is slow. It's actually faster to take the reciprocal of a number and multiply
            // Also note that Division is more accurate than taking the reciprocal and multiplying

#define USE_DIVISION_METHOD
#ifdef USE_FAST_METHOD
            __m128 xRecip = _mm_rcp_ps(x);

            pResultSSE[i] = _mm_mul_ps(xRecip, xSqrt);
#endif //USE_FAST_METHOD
#ifdef USE_DIVISION_METHOD
            pResultSSE[i] = _mm_div_ps(xSqrt, x);

#endif  // USE_DIVISION_METHOD

            // NOTE! Sometimes, the order in which things are done in SSE may seem reversed.
            // When the command above executes, the four floating elements are actually flipped around
            // We have already compensated for that flipping by setting the initial x vector to (4,3,2,1) instead of
(1,2,3,4)

            x = _mm_add_ps(x, xDelta);    // Advance x to the next set of numbers

        }
#endif  // TIME_SSE
#ifndef TIME_SSE
        float xFloat = 1.0f;
        for (int i=0 ; i < length; i++)
        {

            pResult[i] = sqrt(xFloat) / xFloat;    // Even though division is slow, there are no intrinsic functions
like there are in SSE
            xFloat += 1.0f;
        }

#endif  // !TIME_SSE
```

```
    for (int i=0; i < SSELength; i++)
        {

            __m128 xSqrt = _mm_sqrt_ps(x);
            // Note! Division is slow. It's actually faster to take the reciprocal of a number and multiply
            // Also note that Division is more accurate than taking the reciprocal and multiplying

#define USE_DIVISION_METHOD
#ifdef USE_FAST_METHOD
            __m128 xRecip = _mm_rcp_ps(x);

            pResultSSE[i] = _mm_mul_ps(xRecip, xSqrt);
#endif //USE_FAST_METHOD
#ifdef USE_DIVISION_METHOD
            pResultSSE[i] = _mm_div_ps(xSqrt, x);

#endif  // USE_DIVISION_METHOD

            // NOTE! Sometimes, the order in which things are done in SSE may seem reversed.
            // When the command above executes, the four floating elements are actually flipped around
            // We have already compensated for that flipping by setting the initial x vector to (4,3,2,1) instead o
(1,2,3,4)

            x = _mm_add_ps(x, xDelta);    // Advance x to the next set of numbers

        }
#endif  // TIME_SSE
#ifndef TIME_SSE
        float xFloat = 1.0f;
        for (int i=0 ; i < length; i++)
        {

            pResult[i] = sqrt(xFloat) / xFloat;   // Even though division is slow, there are no intrinsic functions
like there are in SSE
            xFloat += 1.0f;
        }

#endif  // !TIME_SSE
```

```
	for (int i=0; i < SSELength; i++)
		{

			__m128 xSqrt = _mm_sqrt_ps(x);
			// Note! Division is slow. It's actually faster to take the reciprocal of a number and multiply
			// Also note that Division is more accurate than taking the reciprocal and multiplying

#define USE_DIVISION_METHOD
#ifdef USE_FAST_METHOD
			__m128 xRecip = _mm_rcp_ps(x);

			pResultSSE[i] = _mm_mul_ps(xRecip, xSqrt);
#endif //USE_FAST_METHOD
#ifdef USE_DIVISION_METHOD
			pResultSSE[i] = _mm_div_ps(xSqrt, x);

#endif	// USE_DIVISION_METHOD

			// NOTE! Sometimes, the order in which things are done in SSE may seem reversed.
			// When the command above executes, the four floating elements are actually flipped around
			// We have already compensated for that flipping by setting the initial x vector to (4,3,2,1) instead o
(1,2,3,4)

			x = _mm_add_ps(x, xDelta);	// Advance x to the next set of numbers

		}
#endif	// TIME_SSE
#ifndef TIME_SSE
		float xFloat = 1.0f;
		for (int i=0 ; i < length; i++)
		{

			pResult[i] = sqrt(xFloat) / xFloat;	// Even though division is slow, there are no intrinsic functions
like there are in SSE
			xFloat += 1.0f;
		}

#endif	// !TIME_SSE
```

Performance difference



ISPC

- A different compiler with some tweaks to standard C
- Produces high-end vectorization code with the user specifying how to vectorize through the foreach clause

ISPC example

```
export void sumInTraceP2(uniform float x, uniform float y,
  uniform float angle, uniform float velocity,
   uniform float input[],float  num[],  float denom[],
     uniform int nt,uniform float dt){

   float t,nn,dd;
   int ishift;
   nn=num[0];
   dd=denom[0];
   t=cos(angle)/velocity*x+sin(angle)/velocity*y;
   ishift=t/dt+.5;
   foreach(it=0 ... nt){
     nn+=input[it+ishift];
     dd+=input[it+ishift]*input[it+ishift];
   }
   num[0]=nn;
   denom[0]=dd;

   }
```

ISPC example

```
export void sumInTraceP2(uniform float x, uniform float y,
  uniform float angle, uniform float velocity,
  uniform float input[],float  num[],  float denom[],
    uniform int nt,uniform float dt){

  float t,nn,dd;
  int ishift;
  nn=num[0];
  dd=denom[0];
  t=cos(angle)/velocity*x+sin(angle)/velocity*y;
  ishift=t/dt+.5;
  foreach(it=0 ... nt){
    nn+=input[it+ishift];
    dd+=input[it+ishift]*input[it+ishift];
  }
  num[0]=nn;
  denom[0]=dd;

  }
```

export-
make available to C

ISPC example

```
export void sumInTraceP2(uniform float x, uniform float y,
  uniform float angle, uniform float velocity,
   uniform float input[],float  num[],  float denom[],
    uniform int nt,uniform float dt){

   float t,nn,dd;
   int ishift;
   nn=num[0];
   dd=denom[0];
   t=cos(angle)/velocity*x+sin(angle)/velocity*y;
   ishift=t/dt+.5;
   foreach(it=0 ... nt){
     nn+=input[it+ishift];
     dd+=input[it+ishift]*input[it+ishift];
   }
   num[0]=nn;
   denom[0]=dd;

   }
```

export-
all vector units
see the same
value

ISPC example

```
export void sumInTraceP2(uniform float x, uniform float y,
  uniform float angle, uniform float velocity,
   uniform float input[],float  num[],  float denom[],
     uniform int nt,uniform float dt){

   float t,nn,dd;
   int ishift;
   nn=num[0];
   dd=denom[0];
   t=cos(angle)/velocity*x+sin(angle)/velocity*y;
   ishift=t/dt+.5;
   foreach(it=0 ... nt){
     nn+=input[it+ishift];
     dd+=input[it+ishift]*input[it+ishift];
   }
   num[0]=nn;
   denom[0]=dd;

   }
```

foreach-
This is the
loop to vectorize