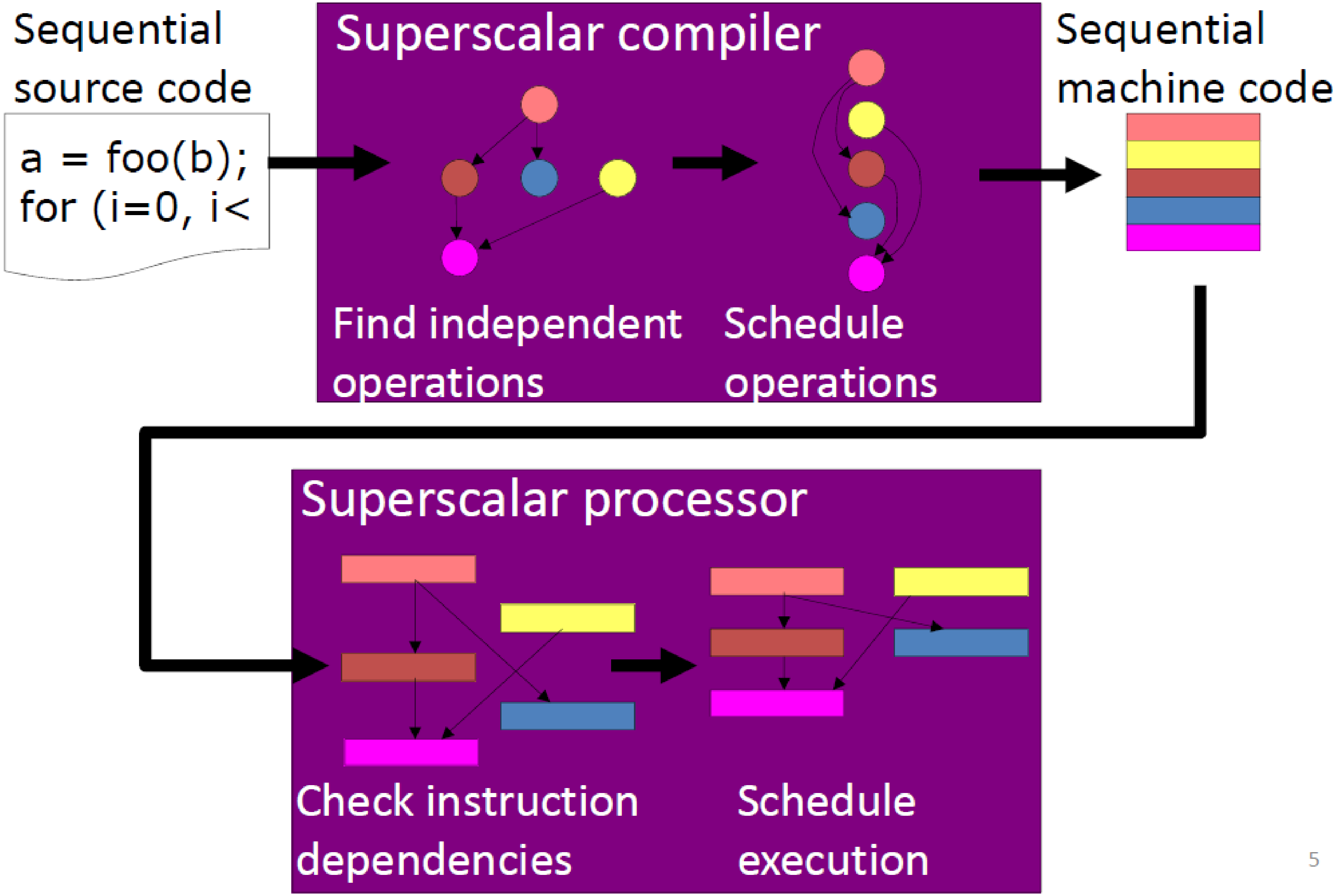
Computação Paralela

Mest. Int. Engenharia Computacional

Ano letivo 2019/2020

Manuel Barroso, Nuno Lau

Superscalar Compiler and Processor



Sequential source code

```
a = foo(b);
for (i=0, i<
```

Superscalar compiler

Find independent operations

Schedule operations

Sequential machine code

Superscalar processor

Check instruction dependencies

Schedule execution

Multimedia extensions

- Extensions to the base ISA that allow a form of vector computation
- “Vectors” are implemented in dedicated registers
  - MMX: 64 bits
  - SSE: 128 bits
  - AVX: 256 bits
  - AVX-512: 512 bits
- Registers of N bits may be used as vectors of 2x(N/2) elements, 4x(N/4) elements, etc.
- One multimedia instruction applies simultaneously to all elements of a register

Multimedia extensions

- MMX
  - 57 instructions added to Pentium
- Extended MMX
- SSE
  - Pentium III
  - 71 instructions (52 FP SIMD, 19 MMX)
- SSE2
  - 144 new instructions (Pentium 4)
  - 128 bit registers
  - Cache-control
- SSE3
  - 13 new instructions (Pentium 4 - 2004)
  - Horizontal processing of values in a register
- SSSE3
  - 32 new instructions (Core)
- SSE4
  - 54 instructions (Penryn - 2008)

Multimedia extensions

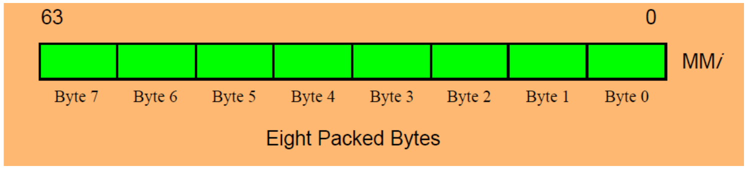
- 3D Now!
  - AMD
  - 45 instructions (21 FP SIMD, 19 MMX, 5 DSP)
- AltiVec
  - Motorola
  - 162 instructions
- AVX
  - Sandy Bridge – 2011
  - 256 bit registers
  - Instructions with 3 operands
- AVX2
  - Haswell New Instructions, 2013
  - Broadcast/permute operations on data elements
  - Vector shift instructions with variable-shift count per data element
  - Instructions to fetch non-contiguous data elements from memory
- AVX-512
  - Proposed in 2013, First Processor 2016
  - 512 bit registers
  - Several extensions: Foundation, Prefetch, Vector Neural Network, etc.

MMX registers

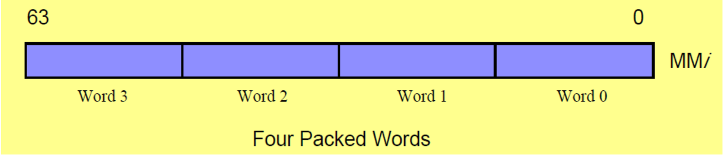
- 8 registers of 64 bits
  - MM0, MM1, …,MM7
  - Are implemented on the same hardware as the FP registers: ST0, …, ST7
    - This allowed to mantain compatibility with existing operating systems

MMX data types

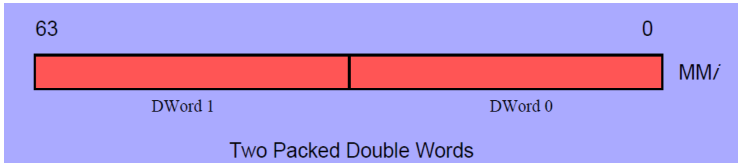
- 8 bytes array



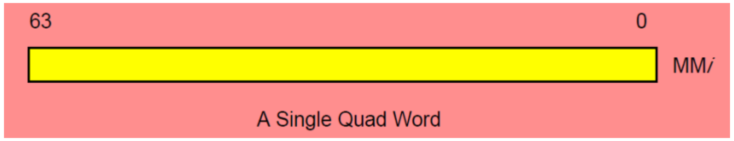
- 4 words array (16 bit/word)



- 2 double words array (32 bits)



- Quadword (64 bits)

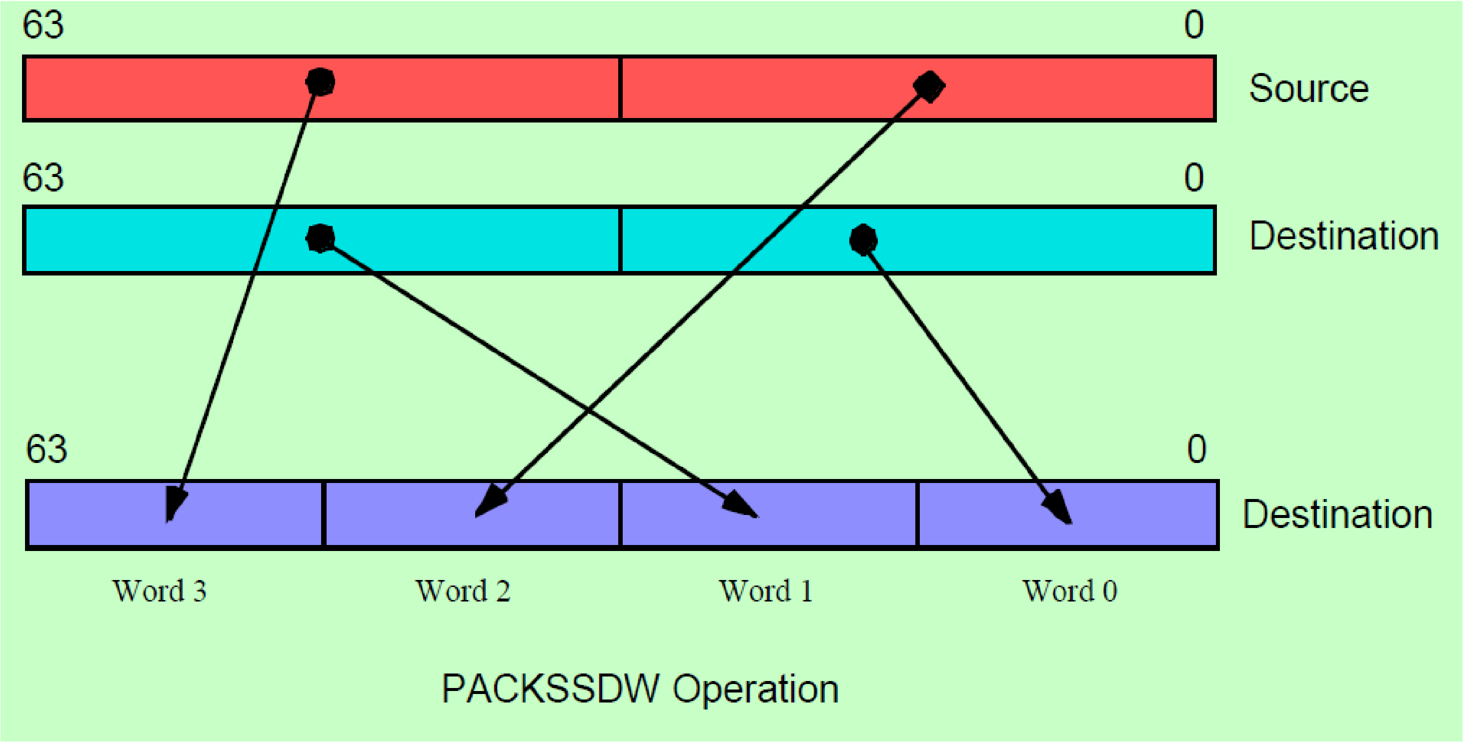


MMX instructions

- Data transfer
  - Move data between registers, memory and MMX registers
  - `movd` (32 bits) and `movq` (64 bits)
- Conversion
  - Converts larger data types to smaller data types and vice-versa
- Packed arithmetic
- Comparisons
- Logic operations
- Shift and Rotate
- EMMS
  - Prepares the processor to execute FP code again

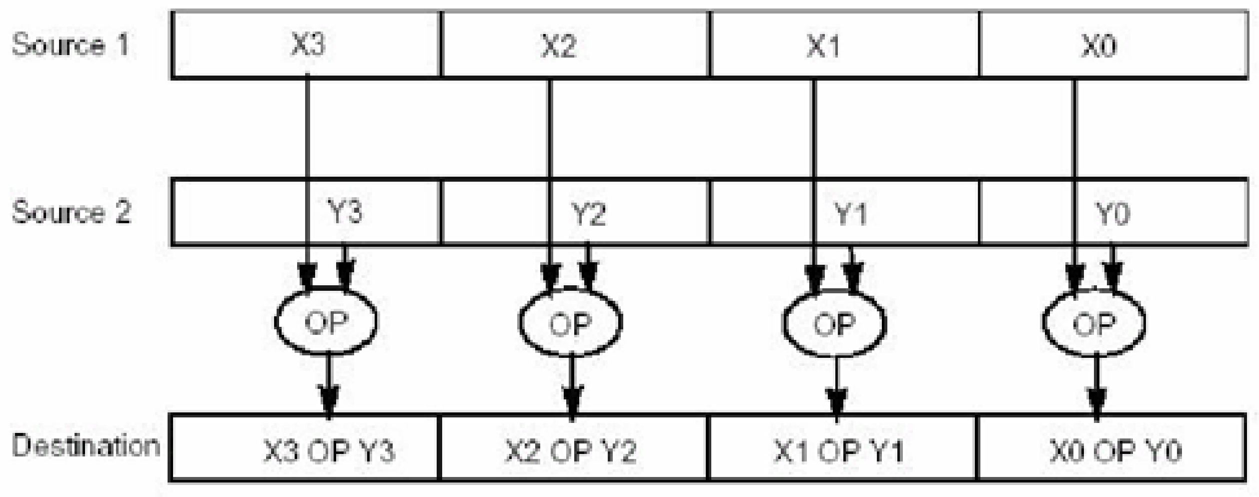
MMX instructions

- Convertion



MMX instructions

- Packed arithmetic
  - SIMD: *Single Instruction Multiple Data*
  - `paddb` (8bit), `paddw` (16bit), `paddd` (32bit)
  - `paddb`, `paddsb` *(signed saturation)*, `paddusb` *(unsigned saturation)*



MMX instructions

Packed Arithmetic	**Wrap Around**	**Signed Sat**	**Unsigned Sat**
Addition	`PADD`	`PADDS`	`PADDUS`
Subtraction	`PSUB`	`PSUBS`	`PSUBUS`
Multiplication	`PMULL/H`		
Multiply & add	`PMADD`		
Shift right Arithmetic	`PSRA`		
Compare	`PCMPcc`		
Conversions	**Regular**	**Signed Sat**	**Unsigned Sat**
Pack		`PACKSS`	`PACKUS`
Unpack	`PUNPCKL/H`		
Logical Operations	**Packed**	**Full 64-bit**	
And		`PAND`	
And not		`PANDN`	
Or		`POR`	
Exclusive or		`PXOR`	
Shift left	`PSLL`	`PSLL`	
Shift right	`PSRL`	`PSRL`	
Transfers and Memory Operations	**32-bit**	**64-bit**	
Register-register move	`MOVD`	`MOVQ`	
Load from memory	`MOVD`	`MOVQ`	
Store to memory	`MOVD`	`MOVQ`	
Miscellaneous			
Empty multimedia state	`EMMS`		

SSE/SSE2 data types

- 8 registers of 128 bits
  - XMM0, …, XMM7
- 4 FPs simple precision (SSE)
- 2 FPs double precision (SSE2)
- 16 bytes (SSE2)
- 8 words (SSE2)
- 4 double words (SSE2)
- 1 inteiro 128bit (SSE2)

Sum arrays

Code:

```
int A[size], B[size], C[size];
…
for (i = 0 ; i < size ; i++)
      C[i] = A[i] + B[i];
```

Parallelism?

Sum arrays

```
movdqa (%eax,%edx,4), %xmm0    # load A[i] to A[i+3]
movdqa (%ebx,%edx,4), %xmm1    # load B[i] to B[i+3]
paddd  %xmm0, %xmm1            # CCCC = AAAA + BBBB
movdqa %xmm1, (%ecx,%edx,4)    # store C[i] to C[i+3]
addl   $4, %edx                # i += 4
```

- movdqa (%eax,%edx,4), %xmm0
  - **mov**: transferência
  - **dq**: double quad
  - **a**: align
  - **(%eax,%edx,4)**: eax + 4*edx

Sum elements of an array

Code:

```
int A[size], total;
…
for (i = 0 ; i < size ; i++)
    total += A[i];
```

Parallelism?

Sum elements of an array

Restructured code:

```
int A[size], temp[4], total;
temp[0]=temp[1]=temp[2]=temp[3]=0;
for (i = 0 ; i < size ; i+=4) {
    temp[0] += A[i];   temp[1] += A[i+1];
    temp[2] += A[i+2]; temp[3] += A[i+3];
}
total = temp[0]+temp[1]+temp[2]+temp[3];
```

Parallelism?

SSE with unrolled loop

- Internal product of 2 vectors (**a** and **b**)

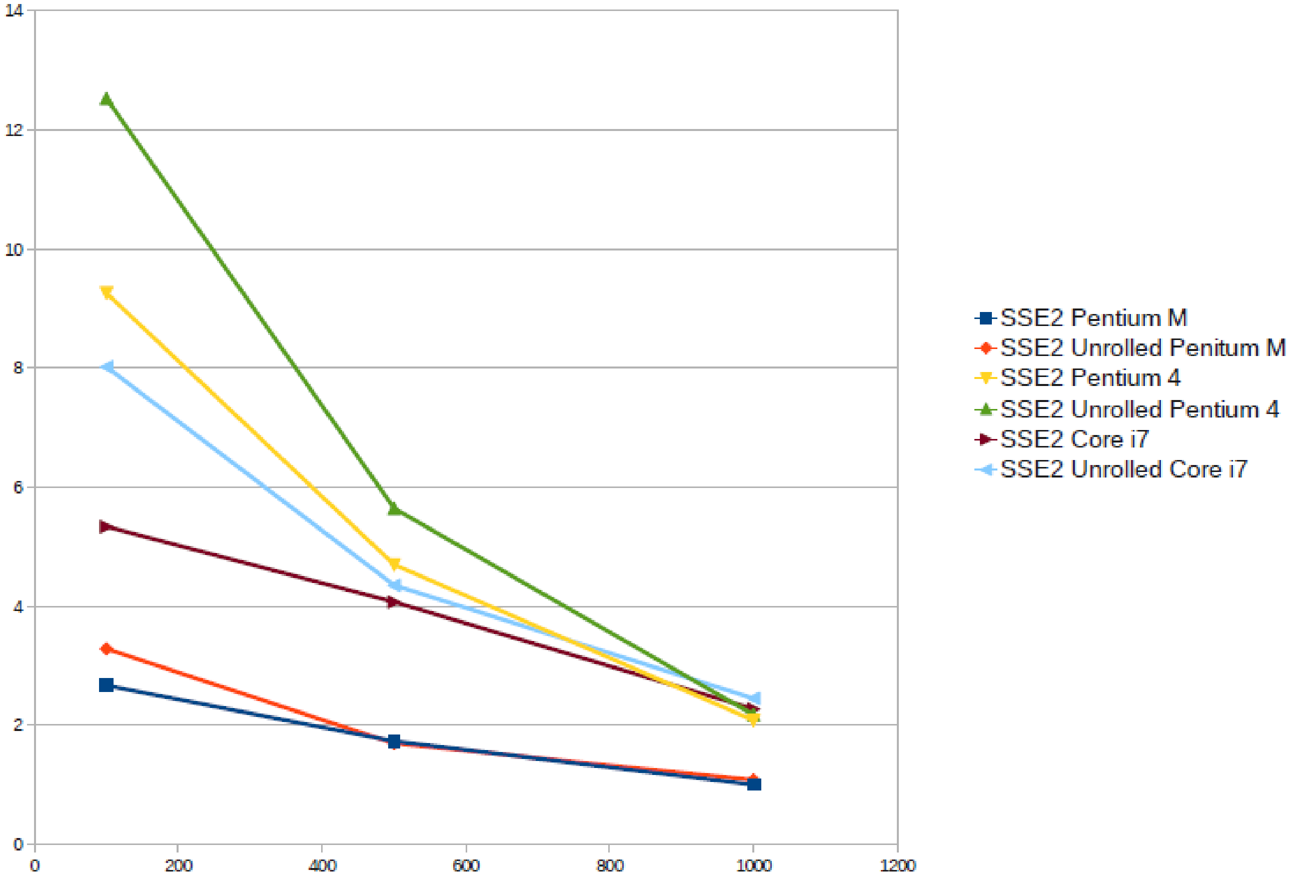
```
length6 = (size/24)*24
for(; i < length6; i += 24){
   __asm__ volatile
   (// instruction comment
    "\n\t movdqa 0x00(%0),%%xmm2 \t#" "\n\t movdqa 0x10(%0),%%xmm3 \t#"
    "\n\t movdqa 0x20(%0),%%xmm4 \t#” "\n\t movdqa 0x30(%0),%%xmm5 \t#"
    "\n\t movdqa 0x40(%0),%%xmm6 \t#” "\n\t movdqa 0x50(%0),%%xmm7 \t#"

    "\n\t mulps 0x00(%1),%%xmm2 \t#”  "\n\t mulps 0x10(%1),%%xmm3 \t#"
    "\n\t mulps 0x20(%1),%%xmm4 \t#”  "\n\t mulps 0x30(%1),%%xmm5 \t#"
    "\n\t mulps 0x40(%1),%%xmm6 \t#”  "\n\t mulps 0x50(%1),%%xmm7 \t#"

    "\n\t addps %%xmm2,%%xmm0 \t#”    "\n\t addps %%xmm3,%%xmm1 \t#"
    "\n\t addps %%xmm4,%%xmm0 \t#”    "\n\t addps %%xmm5,%%xmm0 \t#"
    "\n\t addps %%xmm6,%%xmm1 \t#”    "\n\t addps %%xmm7,%%xmm0 \t#"
   :
   :"r" (a+i), // %0
    "r" (b+i)  // %1
   );
}
```

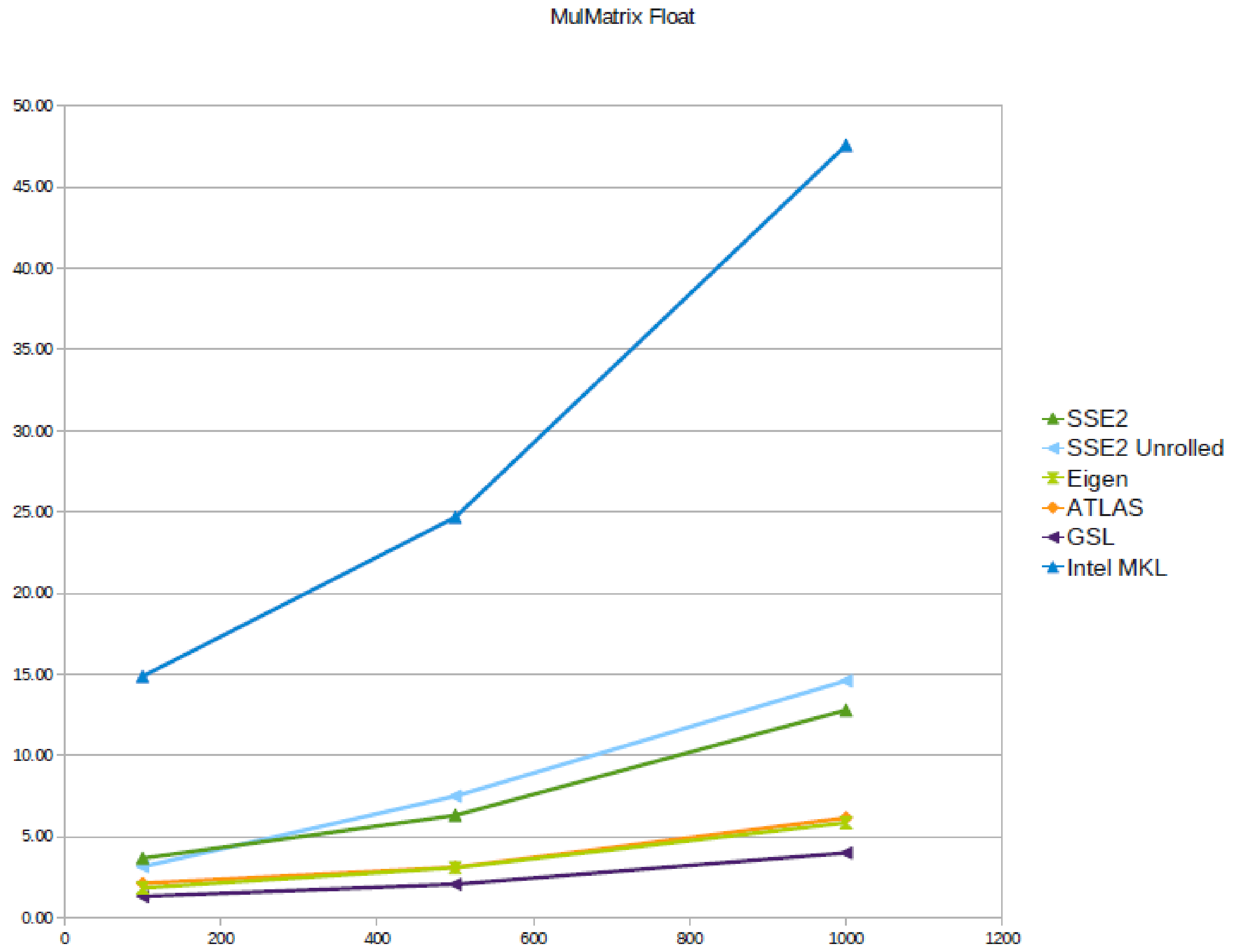
SSE with loop unrolling

- Effect of loop unrolling in internal product



SSE with loop unrolling

- Comparison with other libraries


MulMatrix Float
SSE2
SSE2 Unrolled
Eigen
ATLAS
GSL
Intel MKL