Optimization of Lattice QCD codes for the AMD Opteron processor

Miho Koma (DESY Hamburg)

ACAT2005, DESY Zeuthen, 26 May 2005

▷ **We report the current status of the new Opteron cluster at DESY Hamburg, including benchmarks.**

▷ **We discuss details of the optimization using SSE/SSE2 instructions and the effective use of prefetch instructions.**

[Reference: AMD technical documents # 23932, # 25112]

▷ Lattice QCD simulations

Lattice QCD simulation: Evaluate path integral on a discrete space-time lattice with the Monte Carlo method.

$$\langle \mathcal{O} \rangle = \frac{1}{Z} \int \prod_{x,\mu} dU_\mu \mathcal{O} (detQ)^{N_f} e^{-S_G[U]}$$

"Hot spot" of the numerical calculation:

⇒ **Dirac Operator $Q = \gamma_5 (D + m)$**
⇒ $Q\psi$ is a combination of "Complex 3x3 matrix times complex vector," involves only nearest neighbor interactions.

⇒ **Linear algebra for Spinor Field**
⇒ A spinor field is a vector with $24 \times L^4$ components.
⇒ Major part of the linear algebra is a local operation.

⇒ Easy parallelization
⇒ Suitable for PC cluster

▷ Aim of this work

Develop optimized codes of "Dirac operator" and "Linear algebra" for the AMD Opteron processor.

Step 1:Single processor version (THIS TALK)

Step 2:Parallel processing version

⇒ Our strategy of the optimization can be common to other processors.

Starting point:
Original C code by Giusti, Hölbling, Lüscher, Wittig
Optimized for Intel Pentium 4 (Xeon) processor with SIMD instructions
SIMD: Single-Instruction Multiple data
SSE: Streaming SIMD Extensions instruction sets
SSE register: 128-bit long register for the SSE instruction.

	AMD Opteron	Intel Xeon
Clock Speed	2.4 GHz	1.7 GHz
Cache	L1 (64 kbyte) / L2 (1 Mbyte)	L2 (512 kbyte)
# of SSE/SSE2 Registers	16*	8

*No SSE3

▷ SIMD instructions

A SSE instruction can process packed data (4 floating point numbers / 2 double numbers) on SSE registers. (⇐ Suitable for the lattice simulation)

Example: `addps %%xmm1, %%xmm0`

xmm0:	a	b	c	d
xmm1:	A	B	C	D

xmm0:	a+A	b+B	c+C	d+D

⇒ Max. performance = 4 × CPU clock speed = 9.6 GFlops (single)

We embed the instructions to C codes by defining macro using "Inline assembly"

```
#define _add(in1,in2,out) \
__asm__ __volatile__ (    "movaps %1, %%xmm0 \n\t" \      ! Load data
                          "movaps %2, %%xmm1 \n\t" \
                          "addps %%xmm1, %%xmm0 \n\t" \   ! Calculation
                          "movaps %%xmm0, %0" \           ! Save data
                          :"=m" (out) \
                          :"m" (in1), \
                           "m" (in2))
```

▷ Optimization

Optimization = 1) Reducing the number of instruction = processor independent
2) Hiding the LATENCY time of the processor = processor dependent

Origin of latency: The data is unavailable on time

⇒ **Latency of the SSE instruction itself**

⇒ **Mismatch between the processor speed and the data transfer speed (memory ↔ SSE registers)**

▷ Optimization

Optimization = 1) Reducing the number of instruction = processor independent
2) Hiding the LATENCY time of the processor = processor dependent

Origin of latency: The data is unavailable on time

⇒ **Latency of the SSE instruction itself**

Each instruction needs several processor cycles to finish the operation.

ex.

```
a = b + c   *****          a = b + c   *****
d = e + f    *****         d = a + e        *****
g = h + i     *****        g = d + h             *****
```

CPU has to wait until the previous instruction is finished if the calculations are dependent.

Hiding the latency due to the SSE instruction

$\Longrightarrow$ **Use lower latency instructions if possible**

ex. for sign flip

`mulps` **(5 cycle)** ×

`xorps` **(3 cycle)** ◯

▷ Optimization

Hiding the latency due to the SSE instruction

⟹ **Use lower latency instructions if possible**
ex. for sign flip
`mulps` **(5 cycle) ×**
`xorps` **(3 cycle) ◯**

⟹ **Perform independent operations using several SSE registers**

Make use of 16 SSE registers on Opteron

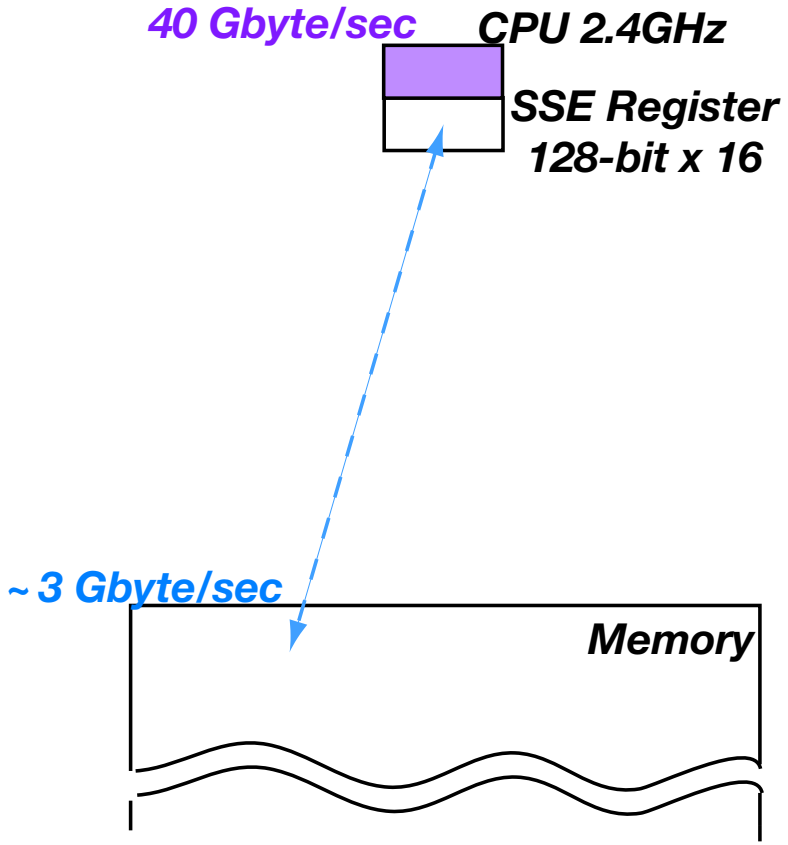
With 2 SSE registers

```
movaps %1, %%xmm0      :**   9 cycles / 4 numbers
movaps %2, %%xmm1      : **
addps  %%xmm1, %%xmm0  :  *****
movaps %%xmm0, %0      :       **
```

With 4 SSE registers

```
movaps %2, %%xmm0      :**  11 cycles / 8 numbers
movaps %3, %%xmm1      : **
movaps %4, %%xmm2      :  **
movaps %5, %%xmm3      :   **
addps  %%xmm1, %%xmm0  :  *****
addps  %%xmm3, %%xmm2  :    *****
movaps %%xmm0, %0      :       **
movaps %%xmm2, %1      :         **
                        ---------------->  Time
```

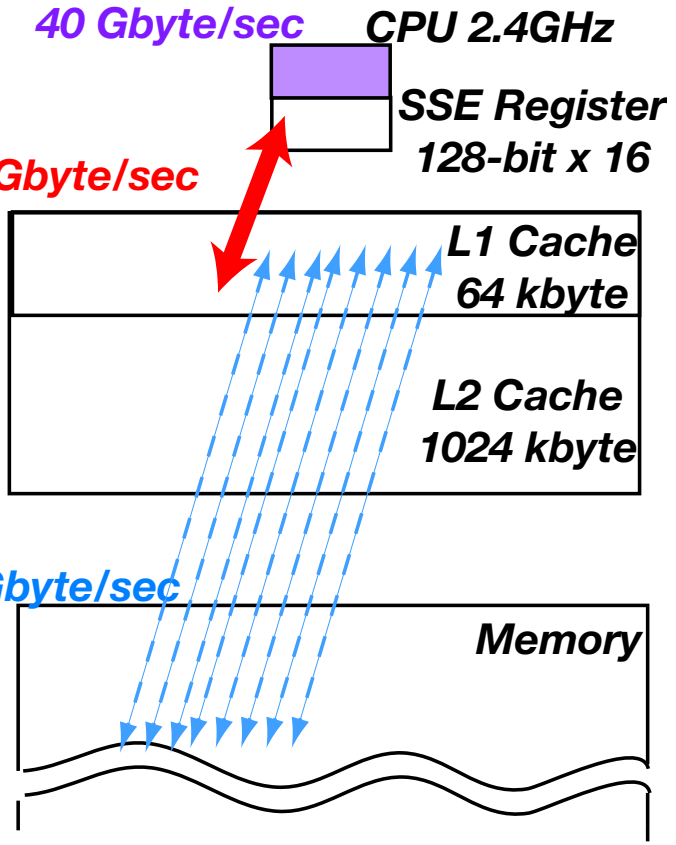
Hiding the latency due to the data transfer speed



Hiding the latency due to the data transfer speed

$\Longrightarrow$ **PREFETCH data to cache**

- **The data transfer speed between the cache and the register is fast enough (2 cycles / 128 bit).**
- **A PREFETCH instruction reads one cache line (=64 byte) from memory into the Level 1 cache.**
- **Data transfer is processed in back ground.**
- **8 PREFETCH instructions can be "in flight" at a time.**
- **The optimal amount of data for each "for" loop $\geq$ 1 cache line**



Important parameter → Prefetch distance

▷ Prefetch distance

Prefetch distance: How far ahead to prefetch
Prefetch distance should be long enough so that the data is in the cache by the time it is needed.

For Dirac Operator Q:
Prefetch distance dependence of the macro for 3x3 matrix times vector

Computation time on $12^3 \cdot 24$ lattice (sec/50 operations)

prefetch dist.	0 (no)	1	2	3	4	5	6
16 SSE registers	2.61	2.13	1.55	1.16	1.49	1.15	1.18
8 SSE registers	2.55	2.16	1.54	1.25	1.52	1.26	1.27

- **Effective use of prefetch can improve performance more than by a factor of 2.**
- **The 16 registers version is faster than the 8 registers version only when the prefetch distance is long enough.**
- **Short prefetch distance $\rightarrow \times$. Long prefetch distance $\rightarrow \triangle$.**

▷ Before and after optimization

Summary of the modification:

- **Use 16 SSE registers if necessary to hide instruction latency**
- **Adjust prefetch distance**

Bench mark calculations:

⇒ **Dirac Operator $\psi = Q\phi$: combination of Complex 3x3 matrix times complex vector per spinor component**

⇒ **Scalar product of two spinors $\langle\psi, \phi\rangle$**

⇒ **Norm square $\langle\psi, \psi\rangle$**

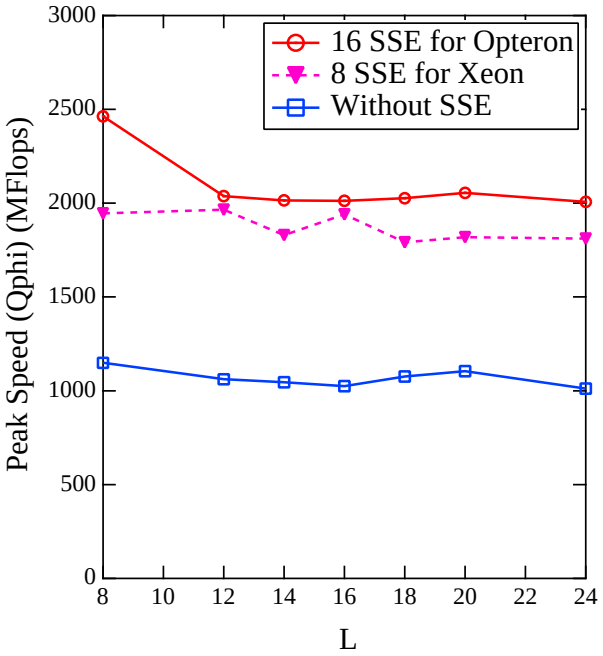
⇒ **Add-assign $\psi = \psi + c\phi$**

- **single precision (32-bit, SSE) / double precision (64-bit, SSE2)**
- **ANSI-C / with SSE (original version for Xeon) / with SSE (new version)**
- **Lattice volume dependence: L=8, 12, 14, 16, 18, 20, 24**

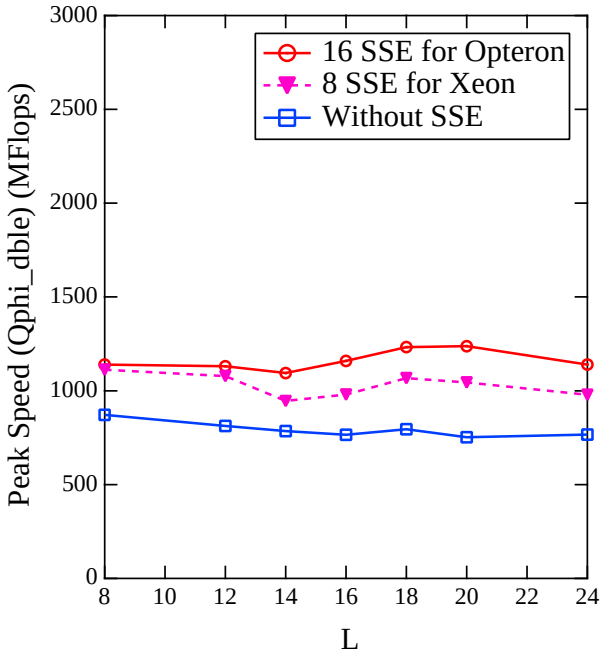
▷ Before and after optimization

Dirac Operator $\psi = Q\phi$:

(Single, 32-bit)



(Double, 64-bit)



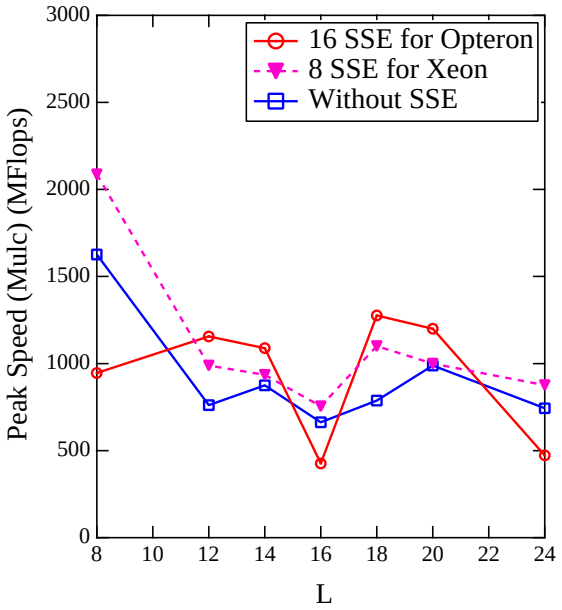
⇒ **For single precision version the code with SSE is twice faster than that without SSE, while the improvement is 20 % for double precision version.**

⇒ **L2 cache improves the performance when the source spinor and gauge fields ($(72+96)*L^4$ byte for single precision) can fit into L2 cache.**

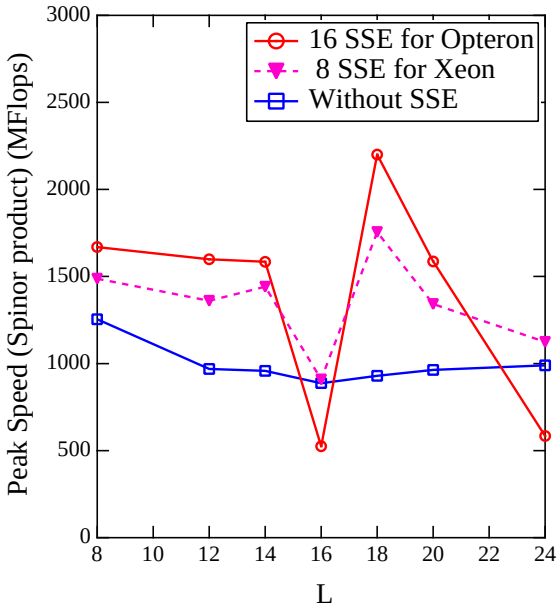
▷ Before and after optimization

Linear algebra (Single, 32-bit)

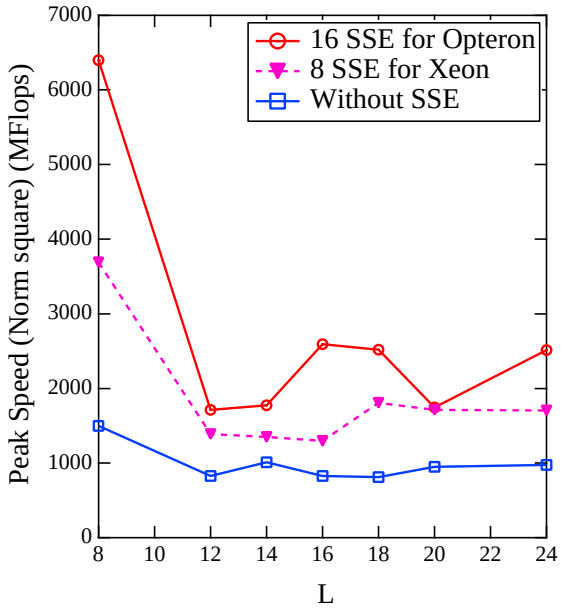
$\psi = \psi + c\phi$



$\langle \psi, \phi \rangle$



$\langle \psi, \psi \rangle$



⟹ **Large L2 cache improves the performance of norm square (6.4 GFlops).**

⟹ **$\psi = \psi + c\phi$ and $\langle \psi, \phi \rangle$ with 16 SSE registers are slower than that without SSE at $L = 8, 16, 24$.**

$\Rightarrow$ Conflict among the multiple prefetch requests ?

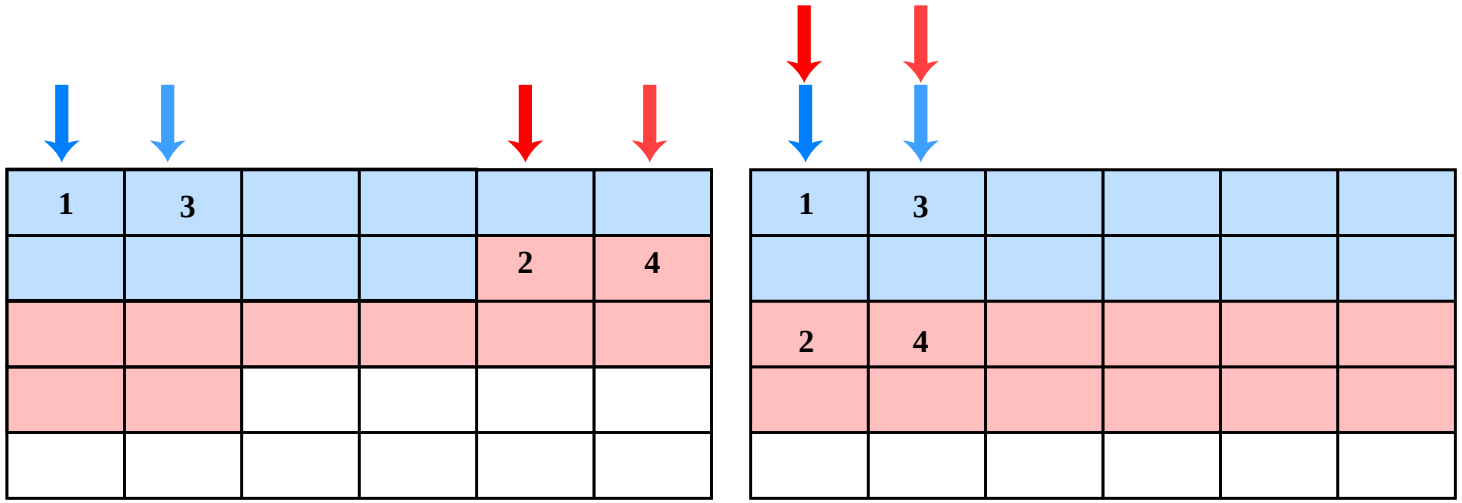
▷ Prefetch (Bank) conflict

Prefetch conflict occurs in the linear algebra calculation when

⇒ **the lattice size is** $(8 * n)^4$

⇒ **two source spinor fields are loaded**

This may happen due to coincidences in the bank structure of the memory system and the data structure of the spinor field.



Bank conflict

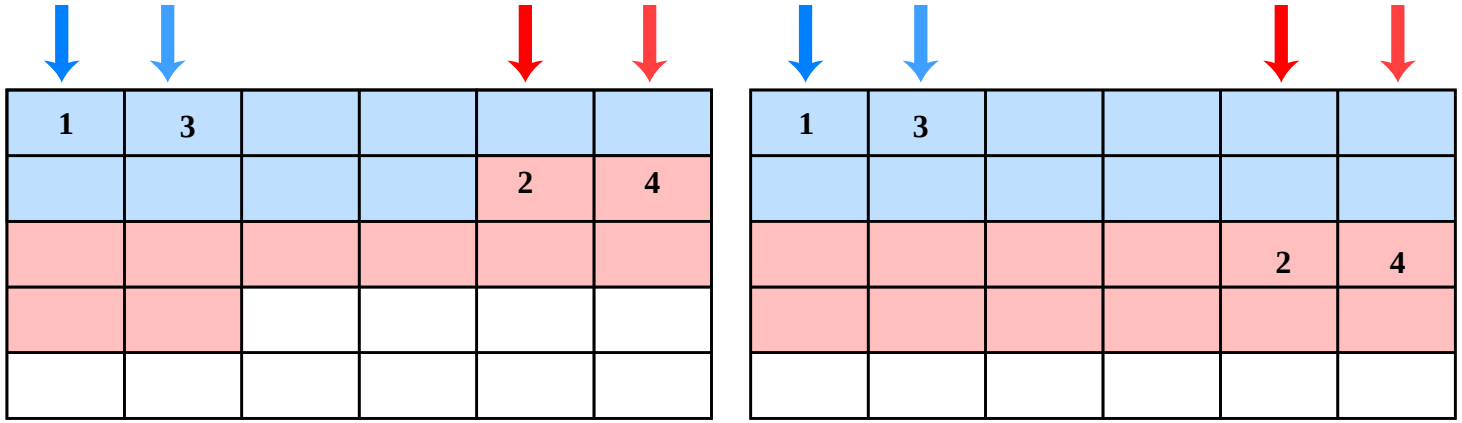
▷ Prefetch (Bank) conflict

Prefetch conflict occurs in the linear algebra calculation when

⇒ **the lattice size is** $(8 * n)^4$

⇒ **two source spinor fields are loaded**

This may happen due to coincidences in the bank structure of the memory system and the data structure of the spinor field.



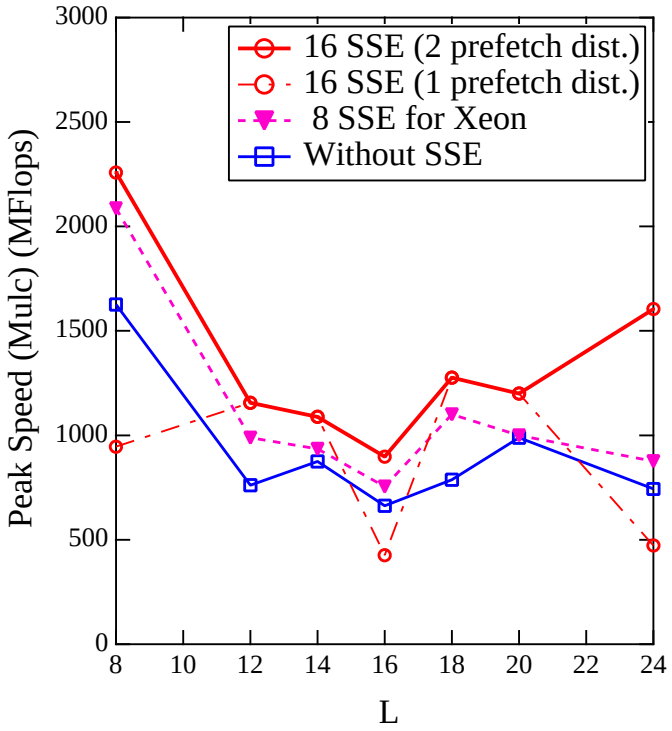
Possible solution: Modify the data structure

Prefetch the second source far more ahead (8 / 2 = 4)

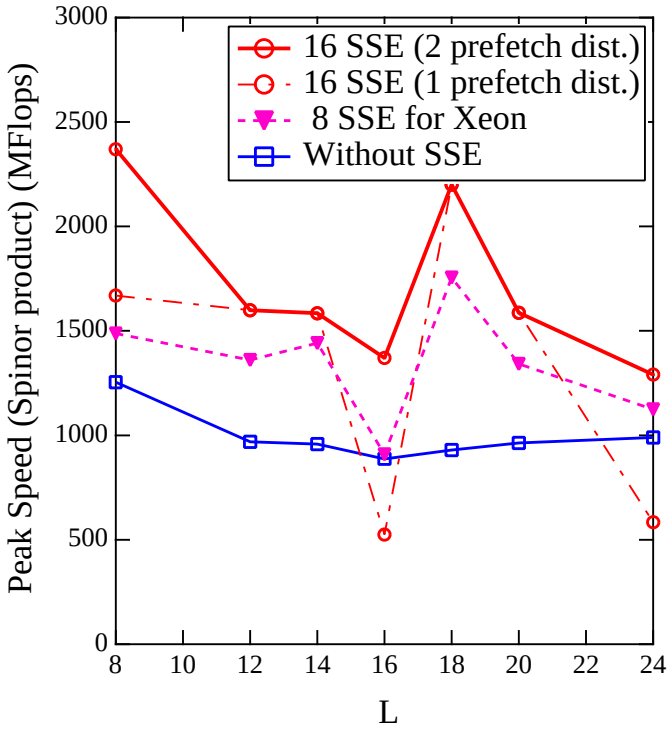
▷ Prefetch (Bank) conflict

Linear algebra

$\psi = \psi + c\phi$



$\langle \psi, \phi \rangle$



⇒ **Adjustment of the prefetch distance cures of the slowing down.**

▷ Pentium Xeon vs. AMD Opteron processor

Peak speed (MFlops)

	Opteron	Xeon	ratio
CPU	2.4 GHz	1.7 GHz	1.41
$\psi = Q\phi$ (single, L=12)	2037	1421	1.43
$\psi = Q\phi$ (double, L=12)	1131	796	1.42
$\langle\psi, \phi\rangle$ (single, L=16)	1370	617	2.22
$\langle\psi, \phi\rangle$ (double, L=16)	661	353	1.87
$\langle\psi, \phi\rangle$ (double, L=12)	1090	554	1.97

$\Longrightarrow$ For $\psi = Q\phi$, the improvement is proportional to the ratio of the CPU clock speed.

$\Longrightarrow$ For linear algebra, we have more gain than expected from the clock speed.
⇒ Cache size
⇒ Data transfer speed

▷ Summary

- ⇒ **We discuss the optimization of the lattice QCD codes for the AMD Opteron processor.**
- ⇒ **Tuning of the prefetch distance can drastically improve performance. (more than by a factor of 2)**
- ⇒ **Bank conflict causes significant slowing down at certain lattice sizes, which can be cured of by adopting two different prefetch distances.**

  The code optimized for the Xeon processor is not the fastest for the Opteron processor. We have considerable gain by tuning prefetch distance and by modifying SSE instructions.

- ⇒ **Large cache effect is observed when the source field(s) can fit into L2 cache.**
- ⇒ **Parallel version of the codes should also be optimized.**
- ⇒ **The strategy of the optimization can be common to other processors.**