



High-level Code Optimization: Case Study with the Wilson-Dirac Operator

By

Wiktor OLKO

LLP ERASMUS PROGRAMME.

Supervised by

Claude TADONKI

Mines ParisTech - CRI

1 The Hopping_Matrix function

Quantum chromodynamics (QCD) [1] is the theory of the strong nuclear force, which is responsible for attracting quarks together to form the nucleons. QCD can be numerically simulated using lattice gauge theory (LQCD) which is formulated on a four dimensional lattice. During LQCD simulation as much as 80% [2] of the time is spent to compute the inversion of the Dirac Operator. It is performed using iterative methods such as Generalized Conjugate Residual or Conjugate Residual. Each iteration consists of computing the Dirac Operator through so-called Hopping_Matrix function.

1.1 Dirac Operator

The input data for the Hopping_Matrix function is a four dimensional lattice of size (L_T, L_X, L_Y, L_Z) , with periodic boundary condition. Number of elements is defined as $VOLUME = L_T \cdot L_X \cdot L_Y \cdot L_Z$. Each vertex of the lattice is a spinor, each edge between two adjacent vertices is a su3 matrix.

A Dirac Operator calculation involves accessing all lattice sites. A calculation on a given site $X \in \mathbb{N}^4: X = [t, x, y, z]$ involves memory accesses to all adjacent spinors and all su3 matrices that are on the edges coming out from the X node. The result is obtained by basic linear algebra operations on the data. The Wilson-Dirac operator can be schematised by Formula 1 [2]. The symbols used in the Formula 1 are explained in Table 1. Values of Dirac gamma matrices are given by:

Formula symbol	Meaning
$U_{x,\mu}$	3x3 complex matrix (su3 matrix)
A	12 x 12 complex matrix
$\psi(x)$	12 components complex vector (spinor)
I_4	Identity matrix of order 4
γ_μ	4 x 4 Dirac gamma matrices
$\hat{\mu}$	μ^{th} vector of canonical basis i.e. $\hat{0} = (0,0,0,1)$, ..., $\hat{3} = (1,0,0,0)$

Table 1: The list of symbols used in Formula 1.

$$\begin{aligned}
 \gamma_0 &= \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix} & \gamma_1 &= \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix} \\
 \gamma_2 &= \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} & \gamma_3 &= \begin{pmatrix} 0 & 0 & -i & 0 \\ 0 & 0 & 0 & i \\ i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix} & \gamma_4 &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}
 \end{aligned}$$

$$D\psi(X) = A\psi(X) - \frac{1}{2} \sum_{\mu=0}^3 \{ [(I_4 - \gamma_\mu) \times U_{X,\mu}] \psi(X + \hat{\mu}) + [(I_4 - \gamma_\mu) \times U_{X-\hat{\mu},\mu}] \psi(X - \hat{\mu}) \}$$

Formula 1

1.2 Data structures

The Table 2 illustrates how Formula 1 variables are represented in C programming language.

Data type declaration	Size [B]
typedef struct { double re,im; } complex;	2 x sizeof(double) = 2 x 8 = 16B
typedef struct { complex c00,c01,c02,c10,c11,c12,c20,c21,c22; } su3;	9 x sizeof(complex) = 9 x 16 = 144B
typedef struct { complex c0,c1,c2; } su3_vector;	3 x sizeof(complex) = 3 x 16B = 48B
typedef struct { su3_vector s0,s1,s2,s3; } spinor;	4 x sizeof(su3_vectors) = 4 x 48B = 192B

Table 2 Data structures used in the Hopping-matrix function (C programming language).

1.3 Memory requirements

The memory requirements can be easily calculated. For each vertex of a lattice one spinor is defined, so there is VOLUME of spinors. The number of spinors is doubled by the output array. For each pair of neighboring nodes there is one su3 matrix. Each node has 8 neighbors (left and right in each of the four directions), so there is $\frac{8 \cdot VOLUME}{2} = 4 \cdot VOLUME$ (Since each edge was counted twice the total number has to be divided by two). Total memory is $2 \cdot sizeof(spinor) \cdot VOLUME + sizeof(su3) \cdot 4 \cdot VOLUME$.

Lattice size	Memory usage [GB]
$32 \cdot 16^3$	0.1 GB
$64 \cdot 32^3$	2.0 GB
$128 \cdot 64^3$	32.2 GB
$256 \cdot 128^3$	515.3 GB

Table 3: Memory requirements for typical lattice sizes.

1.4 Optimization

Ways of optimization will be presented for both single and multi-core processors.

1.4.1 Single core processor

The main algorithm, which consists of a single loop iterating over all lattice sites, has already been highly optimized. However, it is also possible to decrease the number of accesses to the memory and increase processor cache usage by means of reordering the computations or changing the data layout.

1.4.2 Multi-core processor

Ways of improvement for a single core processor are also applicable for multi-core processor, because the calculations on a single core is just a Dirac operator restricted to a sublattice. The new challenge which comes with the multithreaded computing is limited memory bandwidth.

1.5 *The limit for memory access optimization*

Estimation of the limit of memory accesses is essential for determining how much we could improve only by optimizing memory accesses. It is useful for evaluation of algorithms, because it provides the information if it makes sense to search for an optimization even further.

1.5.1 L1, L2 Cache misses

According to Hill [3] there are three types of a read cache misses.

- Compulsory read cache misses.
- Capacity read cache misses.
- Conflict misses.

Table 4 presents details of the classification.

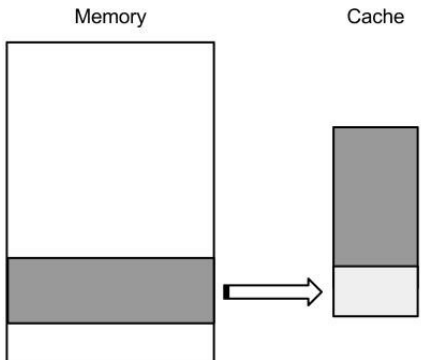
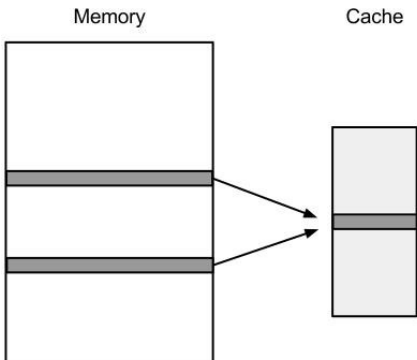
Fundamental reason for occurrence		
First access to data	Limited cache size	
<p>Compulsory read cache miss</p> <ul style="list-style-type: none"> Occurs at first access to data. Cannot be eliminated. The number of compulsory cache misses is a lower bound for all memory accesses to data. 	<p>Capacity read cache miss</p> <ul style="list-style-type: none"> The data in the cache needs to be evicted to make room for the new data, because there is not enough space in the cache. Would not occur for a bigger cache. 	<p>Conflict miss</p> <ul style="list-style-type: none"> The cache size smaller than the memory space enforces using a non-injective function to translate memory address into cache address. Results from an algorithm translating two different memory addresses into the same cache address.
	 <p><i>Illustration 1: The cache has not enough free space to store the memory block.</i></p>	 <p><i>Illustration 2: Two memory locations are mapped to a single cache line.</i></p>

Table 4: Cache miss classification.

As mentioned in the Table 4, number of compulsory cache misses is a lower bound for reading the data from the memory. Table 5 shows how many structures of a given type need to be read from the memory in the main loop of Hopping_Matrix function.

Data type	Number of objects	Size of a single object [B]
su3	4 * VOLUME	144
spinor	VOLUME	192

Table 5: Number of objects read in the Hopping_Matrix function.

1.5.2 32 x 16³ problem size

As specified in the paragraph 1.1, number of elements in a four dimensional lattice of size (L_T, L_X, L_Y, L_Z) is defined as $VOLUME = L_T \cdot L_X \cdot L_Y \cdot L_Z$. For the (32,16,16,16) lattice the

$VOLUME = 32 \cdot 16^3 = 131072$. The number of compulsory read cache misses can be estimated as the total size of data to be processed divided by cache line size. For a 64B cache line there is 1 572 864 L1,L2 compulsory read cache misses (sum of compulsory read cache misses for su3 matrices and spinors).

Data structure name	Compulsory cache misses	Total size of data structure [B]
su3	1 179 648	75 497 472
spinor	393 216	25 165 824

Table 6: Number of compulsory cache misses for the 32×16^3 problem size.

1.5.3 32×16^3 even-odd preconditioning

Even-odd preconditioning is presented in detail in the paragraph 4.2. The main idea is to split the domain into two halves (even and odd lattice nodes), so that the computations for both halves are independent from each other. Below the analysis for only one half is presented.

Despite the fact that only half of the problem (even or odd nodes) is being solved at a time, there is a need to process all su3 matrices data. This is because no two adjacent nodes are present in the same (even or odd) set. For a 64B cache line, half of the spinors and all su3 matrices are accessed, which totals in 1 376 256 L1, L2 compulsory read cache misses (sum of compulsory cache misses for su3 matrices and spinors).

2 Hardware

In this chapter the relevant hardware components will be described and their impact on performance of the computation will be analyzed.

2.1 Processor

Table below presents basic information concerning processors being used for performance measurements.

Machine name	Machine 1	Machine 2	Machine 3
Processor name	Intel Core 2 E8600	Intel Core 2 E8400	pre-launch Intel Ivy Bridge
CPU frequency	3,3 GHz	3 GHz	2.2 GHz
Operating system	Ubuntu 11.10 64bit	Ubuntu 11.10 32bit	RHEL 6
Cache line size	64B		
L1 cache size	32KB for data per core 32KB for instruction per core		
L1 Associativity	8-way associative		

L2 cache	6MB for both data and instruction shared between 2 cores	256KB
L2 Associativity	24-way associative	
L3 cache	-	8MB
Number of cores	2	4
Hyper threading	Enabled	Enabled

2.1.1 Cache design

The cache is constructed of cache line size slots. As mentioned in paragraph 1.5.1 a memory location α needs to be translated to a cache address: slot number and an offset from the beginning of the slot. The following sections will show three algorithm to do the translation. Throughout this section following notation is used: A is the binary representation of α , p is the number of bits needed to enumerate all bytes in a cache line (for a 64B cache line, $p = \log_2(64) = 6$).

Fully associative cache

A is split into two parts: offset $A[0:p-1]$ and tag $A[p:end]$. The algorithm of translating A into a cache address depends on the cache state and eviction policies. If the cache has some empty slots, one of them is used to store the entry. If the cache is full, a cache line is evicted based on eviction policy then the entry is loaded. The offset and the tag are recorded in a lookup table. To find the entry in the cache, all slots need to be checked for a matching tag, which is the biggest disadvantage of the cache design, due to difficulty of implementing it efficiently in hardware [4].

Direct-mapped cache

A is split into three parts: offset $A[0:p-1]$, slot $A[p:p+k-1]$, tag $A[p+k:end]$, where k is the number of bits needed to enumerate all cache slots. The slot is directly chosen based on the slot part of the A address representation. The disadvantage of the cache design is a higher conflict rate [4], because cache eviction may occur when the cache has some empty slots.

N-way associative cache

N-way associative cache is a compromise between a direct-mapped cache and a fully associative cache. Both are not common in modern architectures, the former because of the high conflict rates, the latter because of the difficult implementation in the hardware. It is called a hybrid because it mixes the two approaches. The cache set is chosen in a direct way (bits extracted from the memory address), however, each cache set is a full associative.

A is split into three parts: offset $A[0:p-1]$, set $A[p:p+q-1]$, tag $A[p+q:end]$, where q is the number of bits needed to enumerate all cache sets. The set chunk of an address A is used to directly address the cache set.

L1 data cache consists of $\frac{32KB}{8 \cdot 64B} = 64$ sets.

L2 data cache consists of $\frac{6\text{ MB}}{24 \cdot 64\text{ B}} = 4096$ sets.

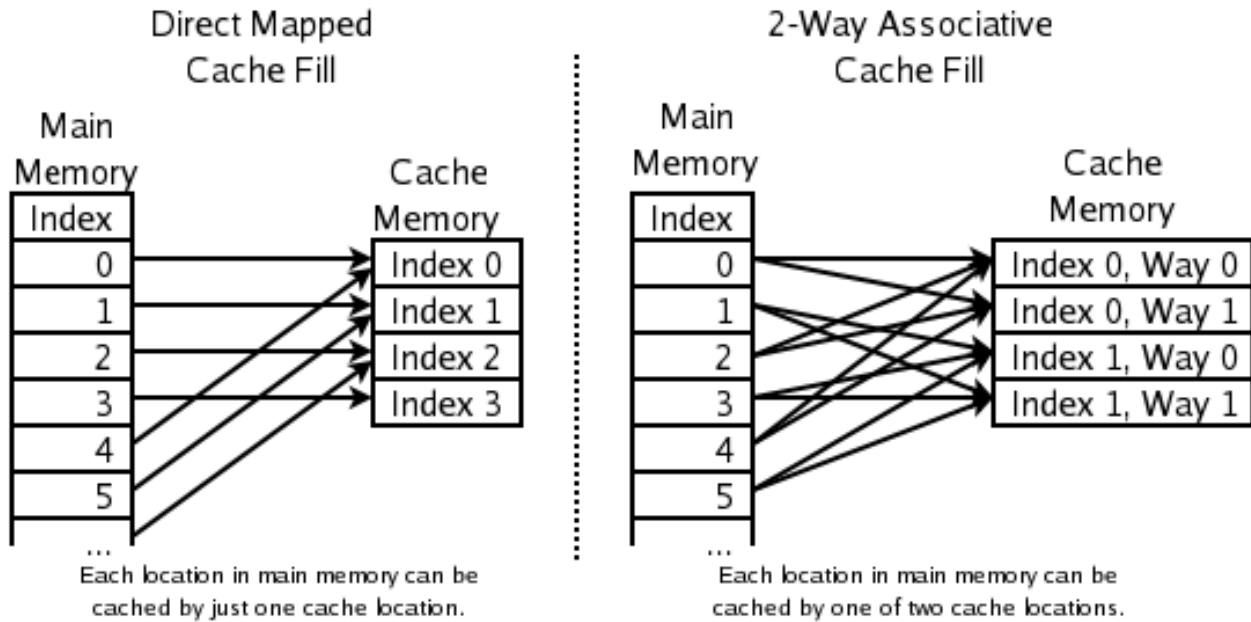


Illustration 3: An example transaction of memory addresses form the main memory to the cache memory for the Direct Mapped Cache and the 2-Way Associative Cache. Source: Wikipedia.

2.2 Memory read latency

L1 data cache read latency differs from processor to processor. The values presented in Table 7 are not precise for Intel Core 2 E8600, they are supposed to give an order of magnitude of the real value.

Type of memory	Number of clock cycles
L1 cache	3 - [5], paragraph 2.2.5.1 [8]
L2 cache	10 - [7]
	14 - [6]
	15 - paragraph 2.2.5.1 [8]
RAM memory	200 - [6]

Table 7 Memory access cost (in clock cycles).

3 Current Hopping_Matrix implementation

The current Hopping_Matrix implementation already consists of multiple optimizations. In the following chapter they will be briefly described. The version is referred in the whole document as the reference version.

3.1 Memory alignment

The data is stored in dynamically allocated arrays. Memory location of each array is shifted, so that it starts at the address which is a multiple of the cache line size. Such alignment minimize the number of memory accesses (cache line loads).

3.2 Tiling

Tiling is a method of splitting the space into smaller parts in order to increase the locality of computations, which results in increasing the cache hit rate. Tiling was implemented to produce four dimensional cubes of size $2^4, 4^4, 8^4, 16^4$, however, no improvement was achieved (There was a significant L2 cache miss rate increase). Following paragraphs explain the reasons for lack of improvement.

3.2.1 Tiling memory requirements

A tile (lattice fragment) size N in a discrete K dimensional space consist of $V = N^K$ nodes. Each node can be represented as a K element vector of the space coordinates $A = (\alpha_0, \dots, \alpha_K), \alpha_{i \in 0 \dots K} \in \{0, \dots, N-1\}$. If the coordinate $\alpha_{i \in 0 \dots K} = 0$ or $\alpha_{i \in 0 \dots K} = N-1$ this means the node is at the tile frontier, so there is a need to access a node from a different tile. The total number of elements accessed outside the tile is equal to the total number of $\alpha_i = 0$ or $\alpha_i = N-1$ in the $A = (\alpha_0, \dots, \alpha_K), \alpha_{i \in 0 \dots K} \in \{0, \dots, N-1\}$ representation of all tile nodes. Since each tile dimension is the same size N , every value of α_i is equally frequent in the $A = (\alpha_0, \dots, \alpha_K), \alpha_{i \in 0 \dots K} \in \{0, \dots, N-1\}$ representation of all tile nodes, so $\alpha_{i \in 0 \dots K} = k, k = 0 \dots N-1$ occurs $\frac{K \cdot V}{N} = \frac{K \cdot N^K}{N} = K \cdot N^{K-1}$ times. Finally, the total number of elements accessed outside the tile is equal to $O = 2 \cdot K \cdot N^{K-1}$, because we count both $\alpha_i = 0$ and $\alpha_i = N-1$.

Example: $N = 4, K = 2, V = N^K = 16$. The grid below depicts A representation of all tile nodes. The border indices are marked in bold.

$a_0 \backslash a_1$	0	1	2	3	...	M-1
0	00	01	02	03		
1	10	11	12	13		
2	20	21	22	23		
3	30	31	32	33		
...	...					
M-1						

3.2.2 32 x 16³ lattice

With even-odd preconditioning only one part “even” or “odd” is considered, so the $K=4$ dimensional space consists of $V = M^4 = 16^4$ nodes instead of $VOLUME = 32 * 16^3 = 2 * 16^4$. Each tile consists of $S = N^4$ nodes, $N \leq M$. Number of elements that need to be accessed from the outside of the tile, to enable computations at the tile frontiers, is equal to $O = 2 \cdot K \cdot N^3 = 8 \cdot N^3$. The space is split into V/S number of tiles.

It is assumed that each tile calculation is independent from each other and the L2 cache is cleared after each tile calculation. Total size in bytes of the tile and the outside spinors is equal to $TS = (S+O) * \text{sizeof}(\text{spinor}) = (S+O) * 192$ [B]. It is also assumed that the tile with the outside nodes fit the L2 cache. Based on this assumption, the theoretical L2 data cache miss rate (TDLmr) is calculated. TL2mr is the sum of L2 cache misses during calculation of each tile and is equal to $TDLmr = V/S * (S + O) * 3$ (cache lines per spinor) + 1 179 648 (su3 obligatory cache misses).

The DLmr value presents the L2 miss rate obtained during experiments using Cachegrind software. The exact values of the parameters described in this paragraph, for a 32x16³ problem are presented in Table 8.

N	S	O	V/S	TS [B]	TDLmr	DLmr
2	16	64	4096	15 360	2 162 688	1 636 593
4	256	512	256	147 456	1 769 472	1 580 335
8	4096	4096	16	1 572 864	1 572 864	1 591 244
16	65536	0 (since N=M)	1	12 582 912	1 376 256	1 470 770

Table 8: Example for the 32×16^3 problem, presenting values described in the paragraph 3.2.2.

3.2.3 32×16^3 lattice conclusions

The theoretical analysis well reflects experiment only for the $N = 8$. For the values $N=2$ and $N=4$ the assumption that the content of L2 cache is evicted after each tile calculation doesn't hold, because L2 cache size (6MB) is many time greater than TS value, so there is no need for evictions and some data are reused. However, for $N=16$ there is an opposite situation, the L2 cache is not capable of holding that much of data. The conclusion can be drawn that the size of the outside spionors to the size of the tile ratio $\frac{O}{S} = \frac{8 \cdot N^3}{N^4} = \frac{8}{N}$, with a L2 cache size 6MB is too high (for small values of N) to be efficient. However, bigger values of N requires a sufficiently large cache size, which are not yet supported by the hardware. This is why Hopping-matrix function does not benefit from tiling.

3.3 $Su3$ matrices double allocation

Each $su3$ matrix is used twice in the Hopping_Matrix function because of the symmetry of the interactions between the adjacent lattice nodes. For convenience of indexing and speed of access each $su3$ matrix is stored twice in the order of the sites accesses. Note that this causes two fold memory overhead. Indexing $u[t+1,x,y,z], u[t-1,x,y,z], \dots, u[t,x,y,z-1]$, which required substantial displacements in the memory was replaced with continuous indexing: $u'[t,x,y,z][0], \dots, u'[t,x,y,z][7]$ which store eight accessed $su3$ matrices in a continuous array block.

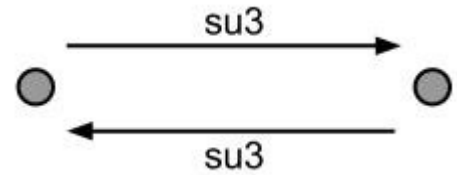


Illustration 4: The symmetry of interactions ($su3$) between lattice nodes (spinors).

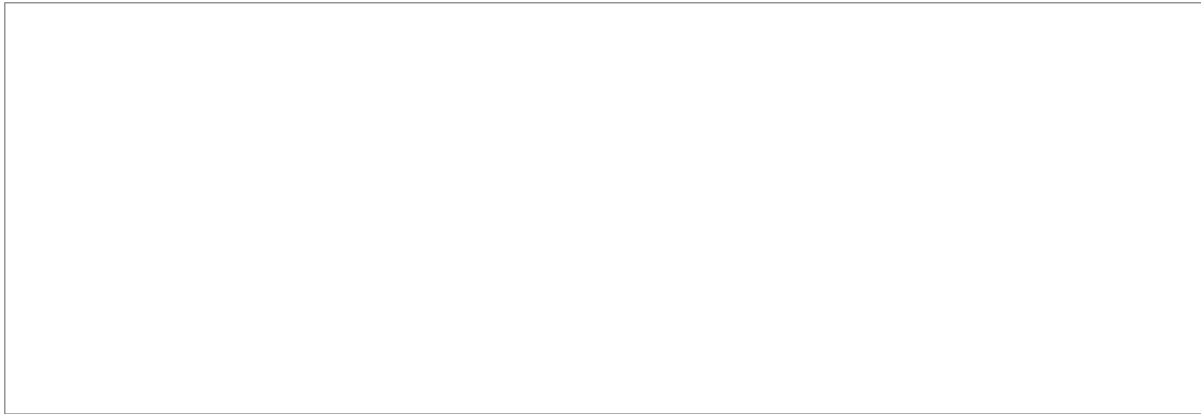


Illustration 5: Replacement of row-major order indexing of su_3 matrices with continuous indexing. For each spinor all 8 adjacent su_3 matrices are stored in a continuous memory block.

3.4 Other optimizations

Optimization	Description
Even-odd preconditioning	paragraph 4.2.
Neighbor indexing	paragraph 4.3.1.

3.5 Multi-core optimization

The reference version can be executed on a multi-core environment. The domain is split into two parts (even and odd), then for each part main loop of Hopping-Matrix function is shared between the cores in an OpenMp style. It is important to choose the right granularity of parallel loops. For example execution of one iteration per thread interchangeably leads to inefficient L1 cache usage.

4 Neighbor spinors access

As mentioned in paragraphs 1.1, 1.3 each iteration of the Hopping_Matrix function consists of basic linear algebra operations on adjacent spinors. In order to access neighbor spinor its array index needs to be obtained. In this chapter methods of calculating indices of adjacent spinors will be discussed.

4.1 Data structure

The four dimensional lattice of spinors is stored in the memory as a one dimensional array. The layout of data is identical as if it was stored in a four dimensional array with row-major order.

Example: $2 \times 2 \times 2$ lattice, row major order, layout in the memory.

(0,0,0)	(0,0,1)	(0,1,0)	(0,1,1)	(1,0,0)	(1,0,1)	(1,1,0)	(1,1,1)
---------	---------	---------	---------	---------	---------	---------	---------

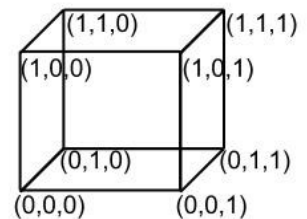
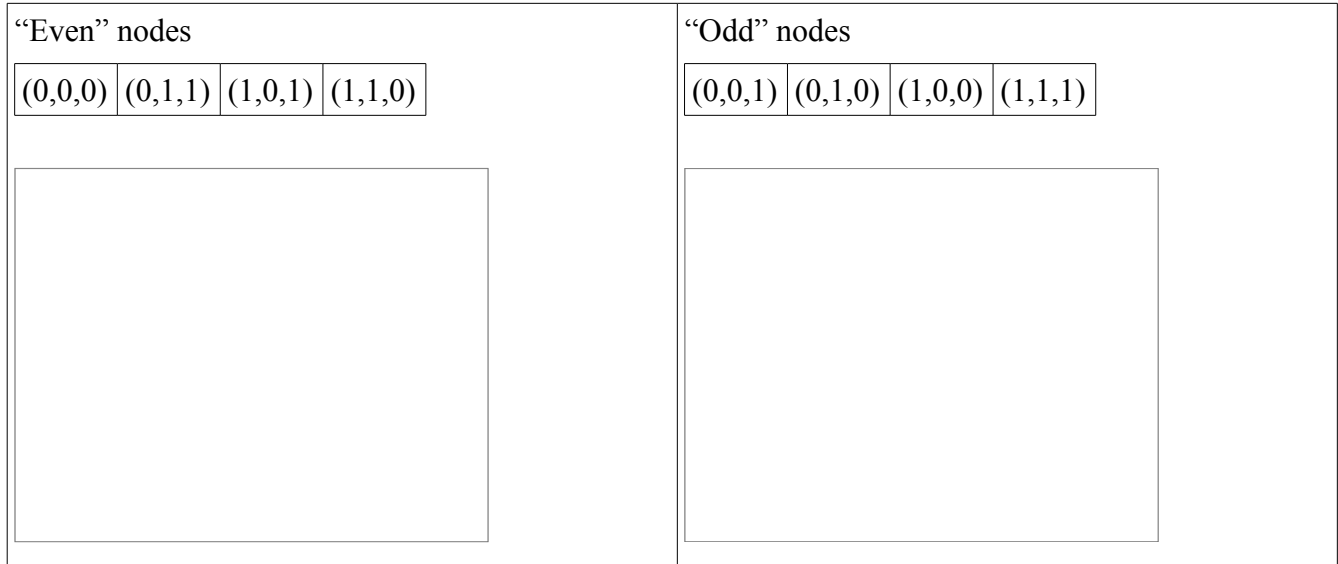


Illustration 6: Visualization of the $2 \times 2 \times 2$ lattice.

4.2 Even-odd preconditioning

A node in N-dimensional space is considered even if the sum of all N coordinates is an even number. Analogously, a node in N-dimensional space is considered odd if the sum of all N coordinates is an odd number. An important property of the even-odd preconditioning is if X is an odd node, then all its neighbors are even and vice-versa. Such an approach has two advantages. Firstly it improves data locality, which may lead to better processor cache performance. Secondly both arrays can be processed in parallel because of no spinor data dependence. The two sets are called “even” and “odd”. The drawback of the method is an inability to reuse the su3 matrices stored in the cache. For odd (and even) nodes each su3 matrix is accessed only once.

Example: 2x2x2 lattice, row major order, data layout of “even” and “odd” matrices.



4.3 Accessing neighbor nodes

There are two ways of accessing neighbor spinors. The first approach is to precompute indices and store them in an array, which doesn't require additional clock cycles for computations during execution of the Hopping-matrix function, but requires an additional memory storage and cache activity. The second is to compute the index on the fly each time it is needed, which doesn't require additional memory storage, but has a computation overhead.

4.3.1 Memory based access

Two arrays of indices are precomputed: *iup* – containing indices of nodes $X(t+1,x,y,z)$, $X(t,x+1,y,z)$, $X(t,x,y+1,z)$, $X(t,x,y,z+1)$ and *idn* – containing indices of nodes $X(t-1,x,y,z)$, $X(t,x-1,y,z)$, $X(t,x,y-1,z)$, $X(t,x,y,z-1)$.

4.3.2 Computation based access

The task of translation lattice indexing into lexicographical is performed using Formula 2.

$$lattice2lexic(t, x, y, z) = z + L_z \cdot y + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot t$$

Formula 2: lattice2lexic function.

For a 32×16^3 problem size the L_z, L_y, L_x values are all powers of 2, which can simplify the computations by introducing shift operations. The products $L_z \cdot y$, $L_z \cdot L_y \cdot x$, $L_z \cdot L_y \cdot L_x \cdot t$ are replaced with shift operations that means respectively $y \ll \log(L_z)$, $x \ll (\log(L_z) + \log(L_y))$, and $t \ll (\log(L_z) + \log(L_y) + \log(L_x))$. The pseudo-code of the `lattice2lexic` function is presented in Listing 1.

```
lattice2lexic(int t, int x, int y, int z)
    return ((t << 12) + (x << 8) + (y << 4) + z)/2;
```

Listing 1: `lattice2lexic` function for 32×16^3 problem size ($\log(L_z) = \log(L_y) = \log(L_x) = 4$).

For a node $X(t, x, y, z)$ eight neighbors are accessed, the way to access them using `lattice2lexic` function is presented in the Table 1. Components such as $(t+1)\%L_T$ or $(t-1+L_T)\%L_T$ in the `lattice2lexic` function calls enable satisfying the periodic boundary condition.

Neighbor indices	The index in the spinors array
$X(t+1, x, y, z)$	<code>lattice2lexic((t+1)%L_T, x, y, z)</code>
$X(t-1, x, y, z)$	<code>lattice2lexic((t-1+L_T)%L_T, x, y, z)</code>
	...
$X(t, x, y, z+1)$	<code>lattice2lexic(t, x, y, (z+1)%L_Z)</code>
$X(t, x, y, z-1)$	<code>lattice2lexic(t, x, y, (z-1+L_Z)%L_Z)</code>

Table 9: Neighbor indices and the way to access them using `lattice2lexic` function.

4.3.3 Binary operations based access

In the binary operations based access the need to create t, x, y, z variables is eliminated. The variables are represented by a single variable icx as a concatenation of t, x, y, z binary representations. This is only possible when L_z, L_y, L_x values are all powers of 2. Each inner variable is accessed by binary product of the icx variable and a precomputed mask. The rest of the algorithm remains similar to 4.3.2, with the respect to division by two, which was eliminated.

4.4 Even-odd lattice traversal methods

As described in the paragraph 4.3.2 the method to access neighbor spinors needs t, x, y, z variables, methods described in 4.3.1 and 4.3.3 do not. The t, x, y, z variables are needed for fast and easy calculation of neighbors. The variable icx is the current spinor index and can be used to access current spinor neighbors through looking up indices in the *iup*, *idn* arrays. Since methods of iterating through “even” or “odd” nodes are equivalent, following paragraphs iterating “odd” will be presented.

Because of the permutation optimization (paragraph 5) the icx variable is obtained from a *loop* array, which maps current loop index to the current spinor index ($icx = loop[i]$). This array of indices can be used to influence the loop scheduling.

4.4.1 Lexicographical approach

The simplest method is to access subsequent spinors as they are stored in the memory and use *icx* index variable and *iup*, *idn* integer arrays to access neighbors indices. This method does not involve calculation of *t*, *x*, *y*, *z* variables, so it cannot be used with computation based access (described in 4.3.2).

```
for i = 0:VOLUME/2
  icx = loop[i]
  [function body]
```

Listing 2: The Lexicographical approach.

4.4.2 Naive approach

Approach presented in the Listing 3 explicitly implements the concept of accessing “even” nodes. Despite its clarity, it involves huge computational overhead because of unnecessary operations (*t,x,y,z* variables incrementation executed in total *VOLUME* instead of $\frac{VOLUME}{2}$ times) and the if statement executed *VOLUME* times.

```
for t=0:LT
  for x=0:LX
    for y=0:LY
      for z=0:LX
        if (t+x+y+z)%2 == 0
          icx++;
          [function body]
```

Listing 3: The naive approach.

4.4.3 Improved approach

Both disadvantages of previous implementation were eliminated due to a trick with the *z* variable initialization and incrementation by two. The pseudo-code is presented in Listing 4. The *t,x,y,z* variables are incremented in total exactly $\frac{VOLUME}{2}$ times and there is no if statement before the function body.

```
for t=0:LT
  for x=0:LX
    for y=0:LY
      for z=(t+x+y)%2:2:LX
        icx++;
        [function body]
```

Listing 4: The improved approach.

4.4.4 Single variable loop

The last approach eliminates the need for four loop condition checking. However, it involves at least two additional calculations to compute t, x, y, z variables. The pseudo-code is presented in Listing 5.

```

for i = 0:VOLUME/2
    icx = loop[i]
    t = (icx & (31 << 11)) >> 11;
    x = (icx & (15 << 7)) >> 7;
    y = (icx & (15 << 3)) >> 3;
    z = (2*icx & 15) + (x+y+t)%2;
[function body]

```

Listing 5: The single variable loop approach implemented for 32×16^3 problem size.

4.5 Experiment

The goal of this experiment is to explain benefits and drawbacks from suggested improvements. All programs were compiled using Icc 13.0.0 and executed on both Machine 1 and Machine 2 (paragraph 2.1). The Ir, D1mr, and DLmr values were obtained using Cachgrind software. Details of the three algorithms tested in the experiment are presented in the Table 10. The execution time presented is the minimal time out of 20 program executions.

Algorithm name	Reference version	Memory based	Computation based
Even-odd lattice traversal methods	Lexicographical	Lexicographical	Improved approach
Accessing neighbor nodes	Inefficient memory based access	Memory based access	Computation based access

Table 10: Details of three algorithms tested in the experiment.

The results of the experiments are presented in the Table 11 and Table 12.

	Reference version	Memory based	Computation based
Instruction reads (Ir)	172 032 038	172 163 089	173 094 749
L1 data read cache misses (D1mr)	2 320 337	2 219 536	2 174 450
L2 data read cache misses (DLmr)	1 470 775	1 424 147	1 388 546
Time Machine 2 [s]	0.0386	0.038	0.0368

Table 11: The results of the experiment on the Machine 2.

	Reference version	Memory based	Computation based
Instruction reads (Ir)	163 643 429	161 808 391	165 568 318
L1 data read cache misses (D1mr)	2 316 673	2 217 057	2 175 490
L2 data read cache misses (DLmr)	1 470 779	1 424 229	1 388 546
Time Machine 1 [s]	0.0341	0.0324	0.0324

Table 12: The results of the experiment on the Machine 1.

4.5.1 Memory usage decrease

The computation based access version is expected to use less memory than the two other algorithms because it does not store the order in which the lattice is traversed (*loop* array of indices) nor does it store the indices of neighbor spinors (*iup*, *idn* arrays of indices). The *loop* array is an array of integers size $\frac{1}{2}\text{VOLUME}$ ($\frac{1}{2}$ because of even-odd preconditioning). The *iup* and *idn* arrays are also arrays of integers, however, the size of each is $4*\frac{1}{2}\text{VOLUME}$ (each of the four directions for half of the problem). The total memory saving is $S = \text{sizeof(int)}*(\frac{1}{2}\text{VOLUME}+2*2*\text{VOLUME}) = 4*4*\frac{1}{2}*\text{VOLUME} = 2*4*\text{VOLUME} = 2'359'296 \text{ B}$. The expected decrease of cache misses is $S / \text{cache line size} = S / 64 = 36864$ L1, L2 data read cache misses. The number can be estimated as the quotient of total size by cache size because of continuous memory layout of the arrays.

Despite the fact that the memory based algorithm is similar to the reference version algorithm it outperformed the reference version algorithm with about 100'000 less D1mr (4% improvement) and about 46'000 less DLmr (3% improvement). The reason for that is a highly inefficient way of accessing neighbor nodes in the reference version. The conversion from lexicographical to even-odd indexing is performed using *g_lexic2eosub* array (*lexic2eosub[iup[ix]]*), which results in multiple cache misses. In the memory based algorithm the *iup* array was in advance translated to even-odd indexing.

As expected the computation based algorithm performed the best in terms of L1 and L2 data read cache misses. In the memory based version there was over 40'000 more D1mr (2.5% increase) and over 35'600 more DLmr (2% increase) than in the computation based. These figures roughly correspond to the calculated figure of 36'864, the small difference might have been caused by some local variables evicted in the computation based version.

4.5.2 Instruction read increase

On the 64B operating system of the Machine 1 there was a noticeable decrease in Ir. As expected the computation based approach generates the biggest number of Ir, however the decrease in L1,L2 cache misses makes it the fastest. The memory approach also had a very good performance on the Machine 1.

5 Permutation optimization

Let us assume that the Hopping_Matrix iteration space is described by an array of indices K. For each index, eight neighbor spinors are accessed. The goal of the permutation optimization is to find a permutation $K' = \text{permute}(K)$, that will maximize the reuse of spinors held in the L1, L2 cache. The

algorithm used in following paragraphs has the even-odd preconditioning enabled. Only “even” nodes are described, because the approach is exactly the same for both “even” or “odd” nodes.



Illustration 7: The 3D lattice visualization with indices of even (red) spinors. It should make understanding of the zigzag strategies easier.

5.1 L1 data cache usage optimization

The number of spinors that could be simultaneously held in a L1 cache depends on the size of the L1 cache, the cache replacement policy and memory organization (see paragraph 6). During each iteration of Hopping_Matrix function, eight spinors are accessed. Every algorithm below, that produces a permutation K' , was designed for a given number of spinors that could be simultaneously held in L1 cache (L1S) and for least recently used (LRU) replacement policy. In each table starting part of the permutation K' is presented. Spinor indices assessed for a second time are highlighted in bold. Next to a value that will be accessed for the second time in a next few iterations, coordinates (x, y) are placed to make clear

...	y+1	y-1	z+1	...
		(-1,2) B		
		(1,1) A		
	B		A	

Table 13: Fragment of a spinor indices table. The value B is first accessed in the y-1 column, 1st row. Then it is accessed in column y+1 and 3rd row. The coordinates (-1,2) indicate the path between the two positions. This way of indication is used throughout paragraph 5.1 to illustrate the reuse of spinors.

where the value is accessed. The idea is depicted by the Table 13.

5.1.1 Lexicographical indexing

Lexicographical indexing is the default indexing K. Following table shows only five out of VOLUME/2 consecutive iterations of Hopping_Matrix function (Index column) and array indices of eight spinors accessed at each iteration (t+1,t-1,...,z+1,z-1 columns). Only one spinor (column z-1) is reused in the consecutive iteration at L1 cache level. LIS = 8.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
0	2048	63488	128	1920	8	120	(1,1) 0	7
1	2049	63489	129	1921	9	121	(1,1) 1	0
2	2050	63490	130	1922	10	122	(1,1) 2	1
3	2051	63491	131	1923	11	123	(1,1) 3	2
4	2052	63492	132	1924	12	124	4	3

Table 14: Spinor indices accessed during five consecutive iterations of Hopping_Matrix function.

5.1.2 Zigzag2D numbering

The easy way of improving lexicographical indexing is reordering K is such a way, that every second iteration access spinors from the next lattice row. Consecutive computations share two spinors (columns z-1 and (y+1 or y-1)). This method is synonymous to 1x1x2x1 (t,x,y,z) tiling on even or odd nodes. LIS = 8.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
0	2048	63488	128	1920	(3,1) 8	120	(-1,1) 0	7
8	2056	63496	136	1928	16	(2,1) 0	(-2,1) 9	8
1	2049	63489	129	1921	(3,1) 9	121	(-1,1) 1	0
9	2057	63497	137	1929	17	(2,1) 1	(-2,1) 10	9
2	2050	63490	130	1922	(3,1) 10	122	(-1,1) 2	1
10	2058	63498	138	1930	18	2	11	10

Table 15: Spinor indices accessed during six consecutive iterations of Hopping_Matrix function.

5.1.3 Improved Zigzag2D numbering

There is a possibility for more efficient reuse of the spinors held in cache. Starting at the 3rd row there is a cycle: 3, 2, 2 accesses to spinors held in the cache. Indices assessed for a second time are highlighted

in bold. The approach is presented in the Table 16. This method is synonymous to $1 \times 1 \times 4 \times 1$ (t,x,y,z) tiling on even or odd nodes. $L1S \approx 18$.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
0	2048	63488	128	1920	(1,1) 8	120	(-1,1) 0	7
16	2064	63504	144	1936	(3,1) 24	(2,1) 8	16	23
8	2056	63496	136	1928	(1,1) 16	(2,1) 0	(-2,1)9	8
24	2072	63512	152	1944	32	(2,1) 16	(-2,1) 25	24
1	2049	63489	129	1921	(1,1) 9	121	(-1,1) 1	0
17	2065	63505	145	1937	(3,1) 25	(2,1) 9	17	16
9	2057	63497	137	1929	(1,1) 17	1	10	9
25	2073	63513	153	1945	33	17	26	25

Table 16 The improved Zigzag2D approach.

5.1.4 Zigzag3D

The Zigzag3D algorithm is based on the same concept as Zigzag2D, however, it makes use of three dimensions as opposed to two dimensions used by Zigzag2D. There is a 4 iterations cycle: 0 or 1 cache hit at the first iteration, then 2, 3, 3 cache hits in subsequent iterations. $L1S \approx 13$. The approach is presented in the Table 17. This method is synonymous to $1 \times 2 \times 2 \times 1$ (t,x,y,z) tiling on even or odd nodes.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
0	2048	63488	(3,2) 128	1920	(3,1) 8	120	(-1,1) 0	7
8	2056	63496	(4,1) 136	1928	16	(-2,2) 0	9	(-4,1) 8
136	2184	63624	264	8	144	(2,1) 128	(-2,1) 136	143
128	2176	63616	256	(4,1) 0	(3,3) 136	248	(-1,3) 129	128
1	2049	63489	129	1921	(3,1) 9	121	(-1,1) 1	0
9	2057	63497	137	1929	17	(-2,2) 1	10	(-4,1) 9
137	2185	63625	265	9	145	(2,1) 129	(-2,1) 137	136
129	2177	63617	257	1	137	249	130	129

Table 17: The Zigzag3D approach.

5.1.5 Zigzag4D

The last algorithm from the ZigzagND series, takes full advantage of the four dimensional structure of spinors, however the LIS is the highest and is approximately equal to 30. This algorithm is the most complicated and difficult to visualize because of its four dimensional nature. The algorithm resembles modification of tiling with cube size of 2^4 and with indexing changed inside the cube. The approach is presented in the Table 18. This method is synonymous to $2 \times 2 \times 2 \times 1$ (t,x,y,z) tiling on even or odd nodes.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
0	(7,4)2048	63488	(3,2) 128	1920	(3,1) 8	120	(-1,1) 0	7
8	(4,3)2056	63496	(4,1) 136	1928	16	(-2,2) 0	9	(-4,1) 8
136	(2,3)2184	63624	264	(-2,3) 8	144	(2,1) 128	(-2,1) 136	143
128	(2,1)2176	63616	256	(-2,1) 0	(-3,3) 136	248	129	(-7,3) 128
2048	4096	0	(3,2) 2176	3968	(2,1) 2056	2168	2049	(-2,1) 2048
2056	4104	8	(5,1) 2184	3976	2064	(-2,2) 2048	(-3,1) 2056	2063
2184	4232	136	2312	2056	2192	(1,1) 2176	2185	(-3,1) 2184
2176	4224	128	2304	2048	2184	2296	2176	2183

Table 18: The Zigzag4D approach.

5.2 Experiment

The experiment was performed on both Machine 1 and Machine 2. The icc 13.0.0 compiler was used. Results are presented in the Table 19, the Table 20, and the Illustration 8. Following observations has been made:

- The main goal of the optimization to reduce D1mr was achieved. Regardless of the machine and the approach to lattice traversal the zigzag4D method performed less L1 data read cache misses than the lexicographical indexing. However, there was also an increase in the DLmr.
- Illustration 8 makes clear that the zigzag2D algorithm performs best in terms of time execution.
- For the Machine 1 the zigzag2D and zigzag3D outperformed the lexicographical indexing in terms of time execution by 10%, however, the exact reasons for improvement are not clear from the data provided.
- The tiling8_zigzag (1st on Machine 2, 4th on Machine 1) is an improved approach to tiling. It splits the space into tiles of size 8^4 , however, the tile itself is iterated using zigzag4D method.

Machine 2	Time [s]	Ir	D1mr	DLmr
Memory based				
lexicographical	0.0374	172,163,089	2,222,241	1,424,150
zigzag2D	0.0371	172,163,089	2,278,000	1,424,146
zigzag2Dnew	0.0377	172,163,089	2,318,625	1,424,146
zigzag3D	0.0371	172,163,089	2,226,336	1,424,163
zigzag4D	0.0371	172,163,089	2,184,384	1,566,370
zigzag4D_new	0.0377	172,163,089	2,151,296	1,562,986
zigzag4D_new2	0.0378	172,163,089	2,392,896	1,555,918
tilling8_zigzag	0.037	172,163,089	2,186,561	1,520,916
tilling8	0.0376	172,163,089	2,264,896	1,520,922
Computation based				
lexicographical	0.0368	173,094,749	2,176,466	1,388,546
zigzag2D	0.0375	173,719,516	2,234,625	1,388,545
zigzag3D	0.038	174,099,788	2,188,419	1,388,547
zigzag4D	0.0382	175,064,176	2,114,050	1,513,326
zigzag4D_mem	0.0383	173,884,526	2,144,260	1,511,080

Table 19: The results of the experiment performed on the Machine 2.

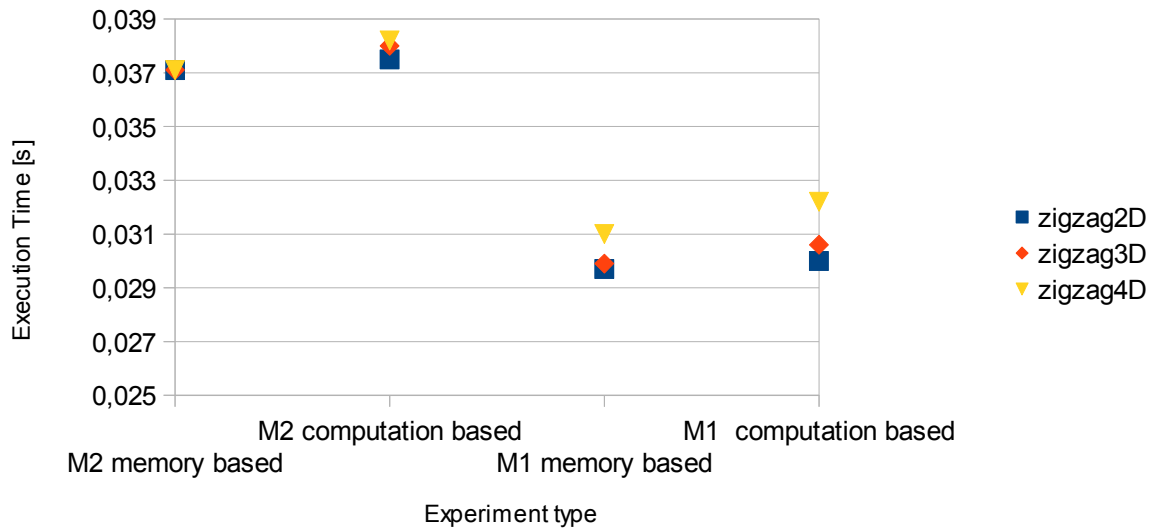


Illustration 8: Execution time as a function of the indexing type, for different machines and indexing strategies. M1 stands for Machine1, M2 stands for Machine2.

Machine 1	Time [s]	Ir	D1mr	DLmr
Memory based				
lexicographical	0.0324	161,808,391	2,215,841	1,424,229
zigzag2D	0.0297	161,808,391	2,276,304	1,424,229
zigzag2Dnew	0.0305	161,808,391	2,318,496	1,424,233
zigzag3D	0.0299	161,808,391	2,226,977	1,424,186
zigzag4D	0.031	161,808,391	2,174,464	1,571,221
zigzag4D_new	0.0314	161,808,391	2,140,672	1,567,723
zigzag4D_new2	0.0317	161,808,391	2,385,216	1,560,343
tilling8_zigzag	0.0303	161,808,391	2,175,488	1,520,814
tilling8	0.0324	161,808,391	2,262,977	1,520,915
Computation based				
lexicographical	0.0323	165,568,318	2,175,490	1,388,546
zigzag2D	0.03	165,737,883	2,235,633	1,388,545
zigzag3D	0.0306	165,967,367	2,187,171	1,388,547
zigzag4D	0.0322	166,998,636	2,110,210	1,519,907
zigzag4D_mem	0.0309	165,819,243	2,136,324	1,518,200

Table 20: The results of the experiment performed on the Machine 1.

5.2.1 Experiment conclusion

No or very little improvement for all zigzag strategies over lexicographical indexing in D1mr is due to insufficient amount of free space in the L1 cache to hold the spinors. This may be caused by the cache pollution from su3 matrices.

5.3 Analysis of lexicographical indexing results

As visualized in Table 14 in lexicographical indexing only one spinor is reused between the two consecutive iterations. However, the analysis using Cachegrind software proved that this solution is more efficient in terms of L1 data cache misses than the Zigzag2D indexing scheme (which shares two spinors between each two consecutive iterations). This fact proves that spinors are retained in cache for far more than just one iteration of Hopping-Matrix function. The goal of this paragraph is to explain the figures of L1 and L2 data cache misses returned by Cachegrind. The analysis will be performed for 32×16^3 problem size, for only even nodes, however, the idea can be applied to other lattice sizes.

It is important to remember that Cachegrind is a very limited model which doesn't take into account events such as hardware/software prefetches and other processes running in parallel on the processor. Table 21 presents main Hopping-Matrix loop indices at which an example node $X(t,x,y,z)$ was accessed ($X(t,x,y,z)$ notation was introduced in paragraph 4.3.1). The node was accessed for the first time when computation of $X(t-1,x,y,z) = X(2,3,3,3)$, $\text{lattice2lexic}(2,3,3,3) = 4506$ were performed. To compute the

value of spinor at position $X(t-1,x,y,z)$ all neighboring su_3 matrices and spinors are accessed, including $X(t,x,y,z)$.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
4506	6553							
6426			6553					
6546					6553			
6553							6553	
6554								6553
6562						6553		
6682				6553				
8602		6553						

Table 21: Presentation of indices where a example node $X(t,x,y,z)=X(3,3,3,3)$, $lattice2lexic(3,3,3,3)=6553$ node was accessed.

5.3.1 Distance between accesses

The distance (the number of Hopping-Matrix loop iterations) between consecutive accesses to the same node can be calculated only for non-boundary nodes, because of periodic boundary condition. Formulas used to compute distances and the example values from Table 21 are presented in Table 22. The distance information is used in following sections to estimate if the spinor data is preserved at given level of cache.

X(t,x,y,z) notation	Decimal index formula.	Example values	Decimal difference between consecutive values (32x16³ problem size).
X(t-1,x,y,z)	$\frac{1}{2}(z + L_z \cdot y + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot (t-1))$	4506	-
X(t,x-1,y,z)	$\frac{1}{2}(z + L_z \cdot y + L_z \cdot L_y \cdot (x-1) + L_z \cdot L_y \cdot L_x \cdot t)$	6426	$\frac{1}{2}(L_z \cdot L_y \cdot L_x - L_z \cdot L_y) = 1920$
X(t,x,y-1,z)	$\frac{1}{2}(z + L_z \cdot (y-1) + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot t)$	6546	$\frac{1}{2}(L_z \cdot L_y - L_y) = 120$
X(t,x,y,z-1)	$\frac{1}{2}((z-1) + L_z \cdot y + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot t)$	6553	$\frac{1}{2}(L_z - 1) = 7$
X(t,x,y,z+1)	$\frac{1}{2}((z+1) + L_z \cdot y + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot t)$	6554	$\frac{1}{2}(1+1) = 1$
X(t,x,y+1,z)	$\frac{1}{2}(z + L_z \cdot (y+1) + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot t)$	6562	$\frac{1}{2}(L_z - 1) = 8$
X(t,x+1,y,z)	$\frac{1}{2}(z + L_z \cdot y + L_z \cdot L_y \cdot (x+1) + L_z \cdot L_y \cdot L_x \cdot t)$	6682	$\frac{1}{2}(L_z \cdot L_y - L_y) = 120$
X(t+1,x,y,z)	$\frac{1}{2}(z + L_z \cdot y + L_z \cdot L_y \cdot x + L_z \cdot L_y \cdot L_x \cdot (t+1))$	8602	$\frac{1}{2}(L_z \cdot L_y \cdot L_x - L_z \cdot L_y) = 1920$

Table 22: Distance between consecutive accesses to the same spinor.

5.3.2 L1 cache result

The values X(t,x,y-1,z), X(t,x,y,z-1), X(t,x,y,z+1), X(t,x,y+1,z) presented in Table 22 are accessed several iterations after each other, so they are preserved in the L1 cache during Hopping-Matrix function execution. The remaining four values X(t-1,x,y,z), X(t,x-1,y,z), X(t,x+1,y,z), X(t+1,x,y,z) needs to be loaded to L1 cache each time prior to computations. This totals in 1+4=5 loads of each spinor to the L1 cache. Since a spinor is 192B it is equal exactly to three 64B cache lines, this makes 15 L1 data cache misses per spinor.

Component contributing to the final value	Number of L2 data read cache misses
Compulsory read cache misses - su3	1'179'648
Spinor L1 cache misses (VOLUME/2 * 15)	995'842
Total sum	2'175'490
The value obtained using Cachegrind software (Lexicographical indexing, computation based access - Table 20)	2'175'490

Table 23: Analytical analysis of the number of L1 data read cache misses.

5.3.3 L2 cache result

During an iteration of the Hopping-Matrix function eight spinors are accessed, one in every direction: $t+1, t-1, \dots, z+1, z-1$. After conversion from lattice indexing to lexicographical, the difference between two extreme indices values (min, max) is no bigger than 4096 (see Table 22). Moreover at all directions $t+1, t-1, \dots, z+1, z-1$ in two consecutive iterations accessed spinors indices are in great majority consecutive. They are not consecutive only at boundaries, but the condition of indices difference being less than 4096 still holds. The first exception are first 2048 iteration of Hopping-Matrix function at the $t-1$ direction, because $X(0-1,x,y,z)$ is transformed to $X(31,x,y,z)$. The second exception are last 2048 iteration of Hopping-Matrix function in $t+1$ direction, because $X(31+1,x,y,z)$ is transformed to $X(0,x,y,z)$ due to the boundary condition.

The 6MB L2 cache is capable of holding 4096 spinors. Since spinors are accessed in more less consecutive order at any given direction, it is needed to load a spinor to from the main memory to L2 cache only once! In vast majority of cases it will be used eight times and evicted from cache. As mentioned above the only situation, when a spinor needs to be loaded for a second time is when it is accessed at $t-1$ or $t+1$ direction in the first 2048 or last 2048 iterations (respectively) of Hopping-Matrix function.

Component contributing to the final value	Number of L2 data read cache misses
Compulsory read cache misses - su3	1'179'648
Compulsory read cache misses - spinors	196'608
Cache misses caused because of first 2048 and last 2048 iterations anomaly	$4'096 * 3 = 12'288$
Total sum	1'388'544
The value obtained using Cachegrind software (Lexicographical indexing, computation based access - Table 20)	1'388'546

Table 24: Analytical analysis of the number of L2 data read cache misses.

5.3.4 Conclusion

The number of L2 cache misses estimated by Cachegrind for lexicographical indexing is extremely efficient and close to optimal. The number of capacity cache misses contributes of less than 10% of all cache misses. The remaining 90% were unavoidable, because this were compulsory cache misses. However, there is room for improvement in terms of L1 data cache misses.

6 Cache optimization

The goal of this optimization is to assure that as much spinors as possible is simultaneously held in the L1 data cache. It is designed for N-way associative L1 data cache with LRU replacement policy.

6.1 Cache pollution

The cache pollution refers to fetching the data into a cache that will override more important, reusable data [9]. Other definition also suggest that the cache pollution occurs when the data loaded to a cache

will not be reused before eviction [10].

During an iteration of the Hopping_Matrix function, eight su3 matrices are loaded. As already mentioned in the paragraph 3.3, none of the su3 matrices are reused in the following computations. Not only the su3 matrices are used only once before eviction, but also they cause an eviction of previous data. What is the most important the spinors, which have the potential to be reused are also evicted from the memory to make room for su3 matrices. It is clear that su3 matrices are causing the cache pollution and this issue needs to be addressed.

6.2 Solution

The solution to tackle the cache pollution is to store su3 matrices in a way, that will minimize the number of cache sets (of a N-way associative cache) used to store the data. The description below uses processor parameters mentioned in paragraph 2.1.

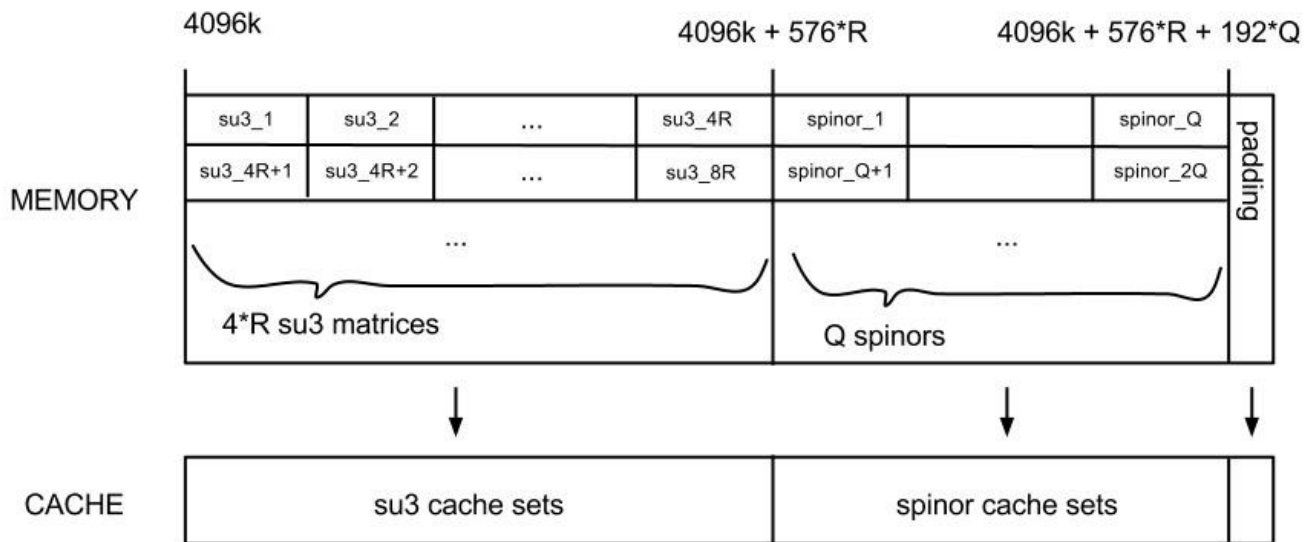


Illustration 9: Continuous memory block shared by su3 matrices and spinors. Projection of memory locations to L1 data cache sets.

To capture the solution, understanding of the way N-way associative cache works is essential. The su3 array is reorganized as follows. Every $4*R$ consecutive su3 matrices are stored at a memory location which address is an multiple of 4096 (page size, the product of the number of cache sets and the cache line size). The efficiency of memory accesses remains almost unchanged, to read 4 consecutive su3 matrices 9 cache reads are needed, however, while iterating through memory skipping unused bytes results in additional computations. What is achieved are su3 matrices at addresses from 0 to $576*R$ (modulo 4096) are always stored in the cache set from 1st to $9*R^{\text{th}}$. The remaining sets are not polluted by su3 matrices.

This solution also require spinors array reorganization. Spinors should use cache sets from $9*R^{\text{th}}$ exclusive. Using the same concept as above Q spinors are stored starting at the address, which quotient remainder by 4096 is equal to $576*R$. Bytes that are not used to store su3 matrices nor spinors (matlab notation) $[576*R+192*Q:4095]$ are left unused. All possible Q values with corresponding $4*R$ values are presented in the Table 25.

4*R	4	8	12	16	20	24
su3 memory pages number	262144	131072	87382	65536	52429	43691
Su3 bytes per page	576	1152	1728	2304	2880	3456
Q	18	15	12	9	6	3
Spinor memory pages number	7282	8739	10923	14564	21846	43691
Spinor bytes per page	3456	2880	2304	1728	1152	576
Memory used	1.07GB	537MB	358MB	268MB	215MB	179MB
Overhead ratio	6,10	3,05	2,03	1,52	1,22	0.01

Table 25: Memory optimization parameters and figures of the memory overhead for a 32×16^3 problem size.

Such memory organization leads to the memory overhead because of two reasons. Firstly the padding is never used, secondly su3 matrices occupy more memory than spinors by six fold. As presented in Table 25 the number of memory pages used by spinors is always smaller or equal than the number used by su3 matrices. The overhead is depicted in the Illustration 10 by the light gray color.

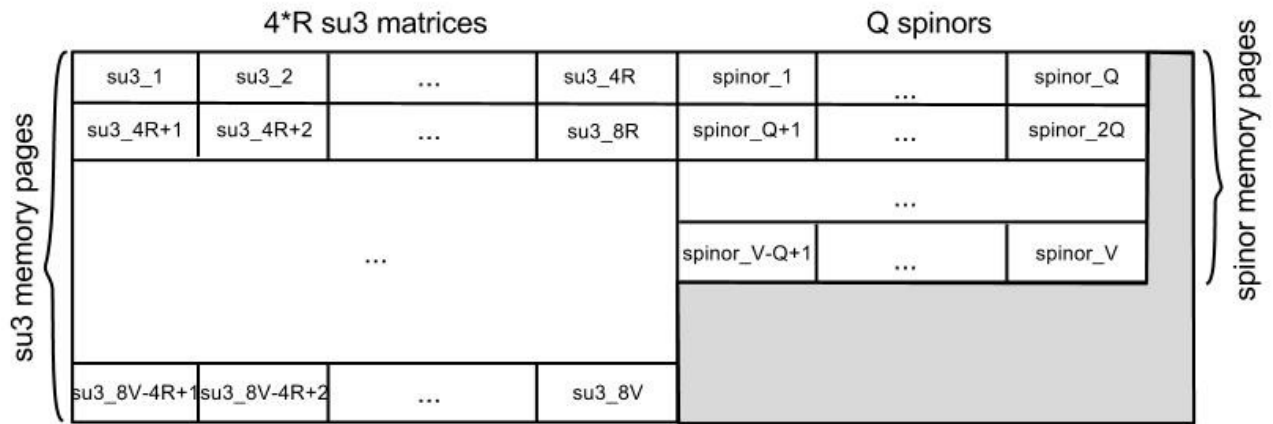


Illustration 10: Number of memory pages occupied by su3 matrices and spinors. The light gray color indicates the memory that is not used.

6.3 Performance

It can be seen at Illustration 10 the smaller memory overhead the faster algorithm. Experiments were performed for $R = 1, 2, 3, 4$ and repeated multiple times on the Machine 2. Different neighbor accessing schemas were tested, however, the trend was not affected.

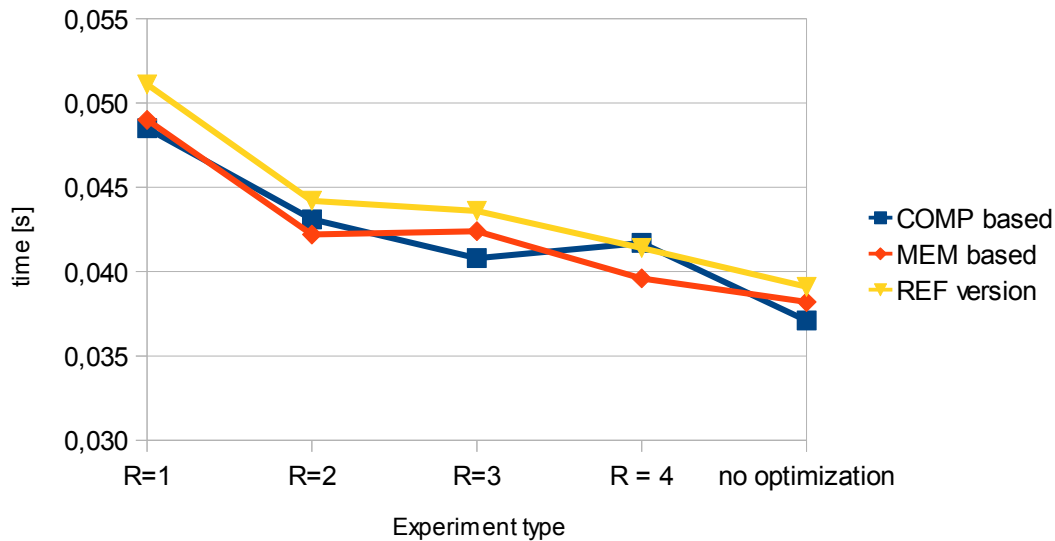


Illustration 11: Time as a function of R in the memory optimized algorithm.

6.4 D1mr experiment

Since the algorithm with $R = 4$ achieved the best performance (Illustration 11), this value was used in the following experiment. The experiment was performed on the Machine 2 using the icc 13.0.0 compiler. The detailed results are presented in Table 26. The D1mr and DLmr values decreased for all of the cases but one. Zigzag2D performed worse which comes as a surprise, probably because of cache pollution caused by the *loop* array. The indexing strategy that benefit most from memory optimization is zigzag4D_new, which improved D1mr by almost 8% and DLmr by almost 9%. However there was no improvement in time execution. The reason for that is substantial increase in number of instructions retired and significant increase in cache misses caused by hardware prefetch.

	Memory optimized			Non-optimized		
	Time [s]	D1mr	DLmr	Time [s]	D1mr	DLmr
lexicographical	0.0393	2 203 619	1 423 914	0.0374	2 222 241	1 424 150
zigzag2D	0.0395	2 409 716	1 423 913	0.0371	2 278 000	1 424 146
zigzag3D	0.0389	2 213 377	1 423 922	0.0371	2 226 336	1 424 163
zigzag4D	0.039	2 023 023	1 423 772	0.0371	2 184 384	1 566 370
zigzag4D_new	0.0393	1 980 677	1 423 769	0.0377	2 151 296	1 562 986

Table 26: Comparison of the memory optimized and non-optimized algorithm results.

6.5 Hardware prefetch

The reason for lack of improvement is due to an active hardware prefetcher which causes multiple L2 cache misses. Effects of hardware prefetches are not included in the Cachegrind software, so to identify the exact reason VTune software was used. Output from the experiment is presented in the Table 27. The most significant factor which decreases the optimization performance is the hardware prefetcher which is a source of huge number of cache misses. Prefetching is discussed in detail in the Paragraph 7.

Parameter description (Core 2 Duo event name)	Zigzag4D, 4*R = 8	Zigzag4D, disabled	Difference [%]
Number of the L2 cache misses (MEM_LOAD_RETIRED.L2_MISS)	687000	646200	6.31
Number of requests from L1 cache to L2 cache (L2_LD.SELF.DEMAND.MESI)	1956000	1863000	4.99
Number of requests from hardware prefetchers to the L2 cache (L2_LD.SELF.DEMAND.MESI)	2485800	2299800	8.09
Number of L2 lines allocated because of demand from the L1 cache (L2_LINES_IN.SELF.DEMAND)	216000	192000	12.5
Number of L2 lines allocated because of demand from hardware prefetchers (L2_LINES_IN.SELF.PREFETCH)	1896000	1524000	24.40

Table 27: Comparison between memory optimization enabled and disabled.

7 Data prefetching

Prefetching data is a technique to load a memory location before it is needed for a computation. The advantage of this technique is that it reduces processor stalls due to computations and load parallelism. However, it has also drawbacks such as possibility of loading data that will never be used or causing eviction of relevant data from cache (cache pollution). In the following paragraph the most important aspects of prefetching are briefly described, then the impact on Hopping-Matrix function is presented.

7.1 Hardware prefetching

The hardware prefetching operates transparently to fetch the data without any programmer effort. There are two areas of the hardware prefetching operation, loading the data from memory to least level cache and loading the data from least level cache to the L1 cache [8].

7.1.1 Least level cache prefetching

In the Core 2 duo E8600 and E8400 processors there are two hardware prefetchers working at the same time using different algorithms to accomplish the task. The first prefetcher is Stremer, it assumes that the data is organized in 128B chunks. When a memory location located in the first chunk half is accessed, Stremer automatically reads the second half (one cache line). The mechanism is very similar

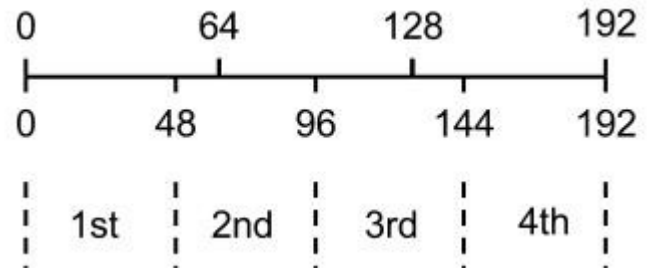
to adjacent cache line prefetch. The second DPL (Data Prefetch Logic) is more advanced. It is triggered by two L2 cache misses which occur inside the same memory page and at the same memory address stride. The address to be prefetched is obtained by adding the stride to the last address being accessed. The DPL is characterized by TTD (Trigger Threshold Distance), the maximal size of the stride in bytes, and by the number of different streams that it track to make prefetch predictions. The Core 2 Duo prefetcher is able to track streams in both forward and backward directions.

7.2 Hardware prefetch impact on Hopping-Matrix

The layout of su3 matrices is the same in every algorithm modification unless specified otherwise. As described in the paragraph 3.3 each su3 matrix in the even-odd preconditioning is accessed only once and all su3 matrices accessed at an iteration are stored in a continuous memory space. This is a perfect situation for the hardware prefetchers, which benefit from continuity of memory layout. However, optimizations proposed in previous paragraphs focus on spinors instead of su3 matrices. Following paragraphs describe impact of prefetching on the performance of the application.

7.2.1 Prefetching spinors

A spinor is a 192B data structure which consists of four su3_vectors (paragraph 1.2). During an iteration of the Hopping-Matrix loop eight spinors are accessed for each direction +0,-0, ..., +3,-3. For +1,-1,+2,-2 directions 1st and 4th su3_vector are first accessed, which results in two cache misses for the [0, 64[and [128,192[bytes. 2nd and 3rd su3_vectors are



accessed second, however, they not result in cache miss, because the data [64, 128[is every 48B. Four su3_vectors marked as 1st, 2nd, 3rd and 4th.

1st and 4th su3_vectors. The computations on +0,-0,+3,-3 directions don't benefit from Stremer prefetches because operation on 1st and 3rd su3_vector requires reading all three cache lines at once.

7.2.2 Prefetching su3 matrix

A su3 matrix is a 144B data structure. Unlike spinors it is loaded all at once, so it doesn't benefit from the Stremer operation. The important thing to mention is an efficient load of su3 matrices occurs only when four consecutive su3 matrices are read before cache eviction. This is because four su3 matrices can be loaded using exactly nine 64B cache lines (only when array containing su3 matrices is 64B aligned).

7.2.3 Reference implementation

The reference implementation strongly benefits from the hardware prefetches. Each column from Table 14 ($t+1, t-1, \dots, z-1$) is represented by a DPL stream, which enables correct prefetches. The prefetcher is capable of tracking 12 different streams in forward direction and 4 in backward direction (paragraph 3.7.2, [8]) which is enough for the case.

7.2.4 Zigzag

Using the analogous reasoning as in paragraph 7.2.3 for the simplest zigzag indexing – Zigzag2D, Table 15 shows there is at least 14 different streams to track for a DPL hardware prefetcher which exceeds its capacity. The analysis for even more complicated Zigzag algorithms shows that the number of stream increases with the complexity of the indexing schema. The zigzag indexing doesn't benefit from DPL prefetcher.

7.3 Software prefetch

Gcc provides extension method `__builtin_prefetch(const void *addr, ...)` with two optional arguments `rw` and `locality`. The value of `addr` is the address of the memory to prefetch. The value of `rw` is compile-time constant one (write) or zero (read). The values of `locality` with corresponding assembly instructions are presented in the Table 28. The `__builtin_prefetch` function reads the address up to a cache line, which totals in maximum of 64B (paragraph 2.5.2 [8]).

Locality value	Assembly instruction	Description
0	<code>prefetchnta</code>	Non-temporal with respect to all cache levels; prefetch data into non-temporal cache structure, with minimal cache pollution. The cache line is loaded from the memory directly to the L1 cache which doesn't results in polluting the L2 cache (paragraph 3.7.2 [8]).
1	<code>prefetcht2</code>	Temporal with respect to the second level cache; prefetch data in all cache levels, except the 0th and the 1st cache levels.
2	<code>prefetcht1</code>	Temporal with respect to first level cache; prefetch data in all cache levels except the 0th cache level.
3	<code>prefetcht0</code>	Temporal data; prefetch data into all cache levels.

Table 28: The locality values with corresponding assembly instructions [11].

7.4 Software prefetch impact on Hopping-Matrix

Several experiments were performed to check if Hopping-Matrix function can benefit from software prefetches. The hardware prefetches during the experiments were not switch off. Software prefetches tend to increase data bus utilization ratio, so it is recommended to use then only when an application is not memory bounded.

7.4.1 Su3 L2 cache pollution

In the Paragraph 6 the ways of addressing the L1 cache pollution were addressed, however, in this paragraph a way to reduce the L2 cache pollution is presented. The idea is to use `prefetchnta` instruction for every `su3` matrix. Since no `su3` matrices are stored in L2 cache it can be used to store even more spinors. Both advantages and disadvantages of such an improvement are presented below.

The experiment proves that the disadvantage of prefetching every `su3` matrix using `prefetchnta` instruction is an increase in requests to the memory, which can be observed as:

- L2 address bus utilization increased by 33%,
- data bus utilization increased by 12%,
- number of bus transaction delays increased by over 70%.

The advantage is reduced number of L2 cache misses:

- Number of L2 data read cache misses reduced up to 30%, however, this doesn't have a huge impact on computations performance because the number of occurrences is relatively low.
- Number of hardware prefetches reduced by 30% and the number of cache misses caused by them reduced by 35%.
- The number of evicted cache lines because of L1 request or prefetch decreased by almost 40%.

Experiments show that the algorithm with prefetched su3 matrices is more than 5% slower than the unoptimized one. It seems that the idea needs further investigation. Nearly 30% reduction of L2 cache misses can lead to significant improvement, however, there is a need to cope with address and data buses utilization ratio increase. One way to do it is to switch off the hardware prefetching and use only software prefetching.

7.4.2 Su3 prefetch strategy

The performance of an application which use software prefetches depends strongly on the locations of `__builtin_prefetch` instructions in an application code. It is a difficult task especially when using `-O3` compiler optimization, which results in rearranging basic blocks to improve code locality [12]. The gain from prefetches is the most significant when a prefetch is done in parallel with computations. A brute force solution was used to check all possible locations in the program code where to place prefetch instructions. The su3 matrix was prefetched 4 su3 matrix loads in advance. It turned out that there was no improvement because of software prefetches.

7.4.3 Lexicographical spinor prefetch strategy

As specified in section 5.3.3, most of the spinors are loaded to L2 cache only once. So the simplest strategy is to prefetch the $t+1$ direction value, because it is the biggest for the vast majority of Hopping-Matrix function iterations (with exception to last 2048 iterations) and this memory location will be reused when accessed at different directions. Such prefetch strategy can lead up to 5% time execution improvement only when the run on a single core, which means the application is not memory bounded. Otherwise additional memory

8 Hybrid algorithm

A common approach to improving algorithms is to merge two different solutions into one, so that the final algorithm benefit from both. The following paragraph will present such algorithms and their experimental results.

8.1 Improved Zigzag2D split into halves

The improved Zigzag2D algorithm was presented in paragraph 5.1.3. The idea to modify the algorithm

stems from the layout of reused spinors indices. As presented in Table 16 only values in the $y+1$, $y-1$, $z-1$ columns (2nd half) were accessed from the L1 cache. Values in the $t+1$, $t-1$, $x+1$, $x-1$ columns (1st half) did not benefit from the L1 cache. The solution is to calculate the two parts separately, each iterated using Improved Zigzag2D, to benefit more from the L1 cache reuse. The final score is obtained by summing up the result from both halves. The performance of reading su3 matrices should not decrease since 4 continuous su3 matrices are read. The layout of spinor indices accessed in the first several iterations is presented in Table 29. The thin column in the middle of the table separate the halves.

Index	t+1	t-1	x+1	x-1	y+1	y-1	z+1	z-1
0	2048	63488	128	1920	8	120	0	7
16	6144	2048	4224	6016	24	8	16	23
8	4224	128	2304	2048	16	0	9	8
24	8320	4224	6400	6144	32	16	25	24
1	2304	63744	384	128	9	121	1	0
17	6400	2304	4480	4224	25	9	17	16
9	4480	384	2560	2304	17	1	10	9
25	8576	4480	6656	6400	33	17	26	25

Table 29: Indices of spinors accessed in the first few iterations.

The algorithm was tested on both Machine 1 and Machine 2 using both icc and gcc compilers. The comparison with the original version is performed for the fastest out of four, which is icc compiler executed on Machine 1. As expected the number of L1 data read cache misses decreased by 6,5%, however, the original version was almost twice faster. The main reason for such a slow performance is a significant increase in the L2 cache misses (over 21%), which leads to the memory bus usage increase. Full experiment results are presented in Table 30.

Machine	Compiler	Time [s]	Ir	D1mr	DLmr
Machine 1	Gcc 4.6.1	0.0533	177,668,110	2,169,680	1,807,865
Machine 1	Icc 13.0.0	0.0529	163,184,649	2,167,566	1,807,848
Machine 2	Gcc 4.6.1	0.0668	172,228,626	2,176,776	1,807,892
Machine 2	Icc 13.0.0	0.0598	172,228,626	2,175,524	1,807,892

Table 30: Experimental results for modified Zigzag2D algorithm.

9 Even-odd su3 cache level sharing

The proposed algorithm is designed for a multi-core processor. The most significant difference comparing to the reference version implementation is resignation from double su3 allocation. Thanks to

this property even and odd parts executed in parallel can reuse common su3 matrices using a cache shared between the cores. All experiments presented in this paragraph were performed on the full 32×16^3 lattice.

9.1 Reaching memory limits

The optimizations described in paragraphs 4, 5, and 7 resulted in a single threaded implementation faster by 10% from the reference version implementation. However, when executed on two cores the efficiency for 2 cores was equal to 0.73 on both Machine 1 and Machine 2. The execution time for a given number of cores is presented in the Chart 1.

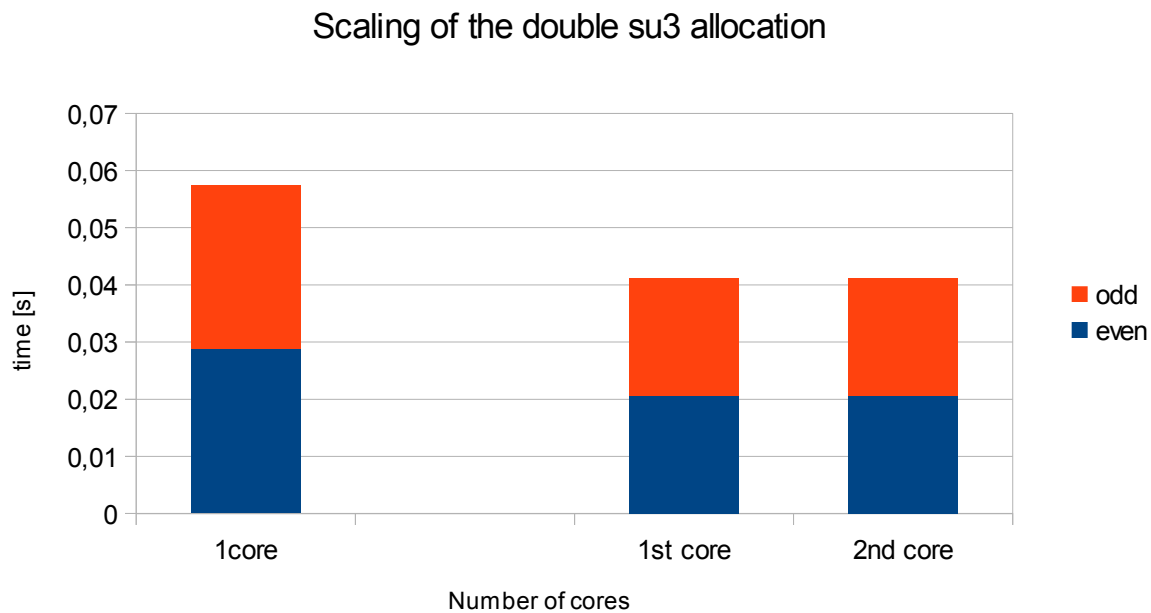


Chart 1: Chart depicting scaling of the double su3 allocation algorithms on Machine 1 and 2.

This suggested that the Hopping-Matrix function is memory bounded. Due to limited bandwidth of the memory bus, the processor stalled waiting for the data from the memory. To prove this point two applications were created. The first was supposed to determine how much time is spent by the Hopping-Matrix function on memory accesses, the second to determine how much time is spent on computations. Comparing these results makes it possible to determine whether the main problem is CPU or memory bounded. To achieve the first goal, almost all computations from the Hopping-Matrix function were commented out, only a few were left to prevent compiler from removing the dead code. To achieve the second goal all computations were performed on the same data, so there was no need to load it from the main memory. Both implementations were thoroughly tested using Cachegrind profiler to verify if relevant parameters (the number of instructions or the number of memory reads) were exactly the same as in original implementation or as designed, equal to zero.

Scaling of the CPU and the memory on multiple cores

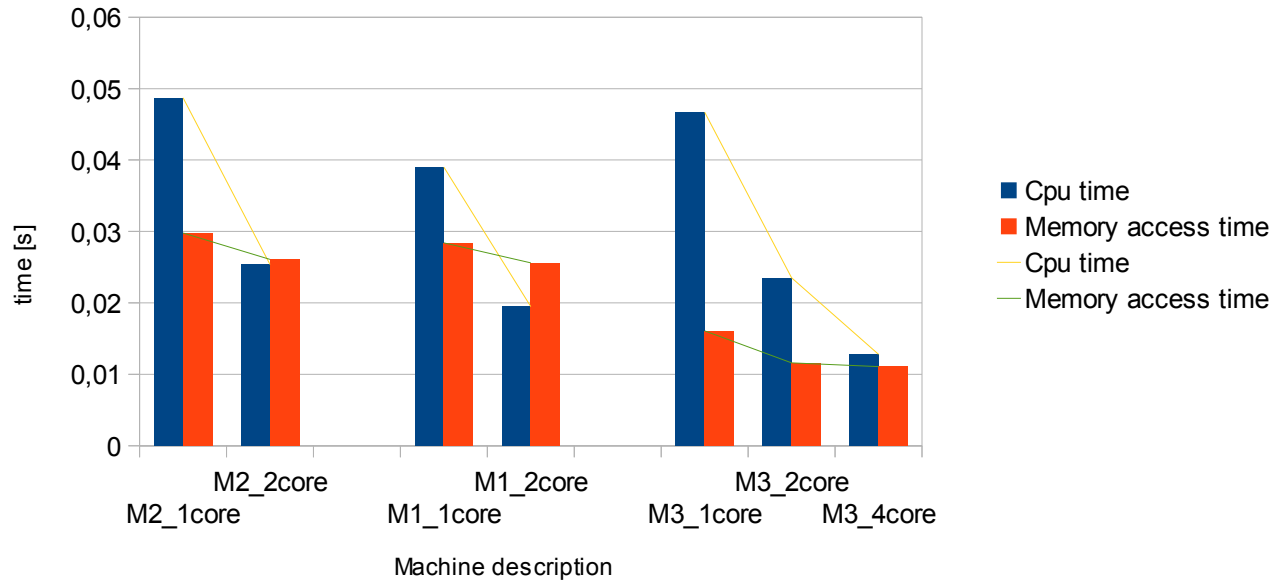


Chart 2: Execution time of two applications (blue - CPU, red - memory) designed to measure the time spent loading memory and the time spent on computations in Hopping-Matrix function. Test were performed on three different multi-core machines.

Data presented in Chart 2 proves, that on both Machine 1 and Machine 2, the Hopping-Matrix function is memory bounded. Machine 3 has more than twice greater memory access speed than the other machines, however, Machine 3 processor is just 2.2GHz, comparing to 3GHz (Machine 2) and 3.33GHz (Machine 1). This is the reason why Hopping-Matrix function on Machine 3 is CPU bounded. All Ivy Bridge Core i5 and Core i7 processors which are publicly available have significantly higher frequency, this is why ways of improving CPU time will not be presented in this paragraph.

9.2 Overcoming memory limits

Because the memory limit has been reached, the only way to improve performance of Hopping-Matrix function was to reduce the amount memory loaded. Decision has been made to resign from the double su3 allocation and try to reuse the su3 matrices at all cache levels.

9.2.1 Consequences of single su3 allocation

The consequence of the single su3 allocation is a significant increase in the hardware prefetcher cache pollution and the number of cache misses. This is because of highly irregular access patterns. Chart 3 illustrates how severe is this penalty. “Su3 cache level sharing” solution executed on a single core is highly ineffective comparing to “double su3 allocation” despite the fact that the same amount of memory is loaded. The only reason of slowdown is the order of reading memory locations.

Comparison of memory scaling on different machines.

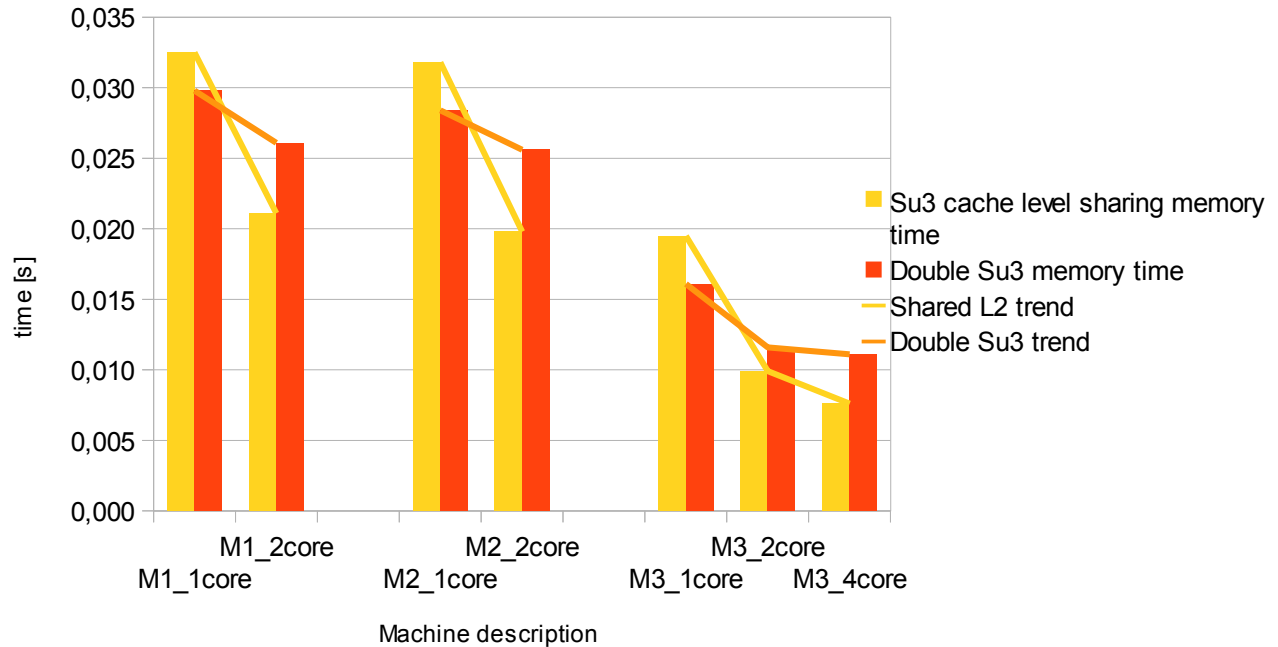


Chart 3: Comparison of reference version (with double su3 allocation) with Su3 cache level sharing solution. Both implementation measure just the memory time.

9.2.2 Su3 cache level sharing

The new approach to Hopping-Matrix function is to schedule even and odd part on one core each and start both threads exactly at the same time. Both threads are not synchronized in any way during execution. Because of exactly the same memory access patterns and exactly the same amount of computations it is assumed that the threads will execute at the same pace.

Since two adjacent spinors (one even, one odd) share a su3 matrix, when both cores execute at nearly the same pace, the su3 matrix will be loaded by one of the cores, and then reused by the other. Such memory optimization leads to better memory scaling (Chart 3), which eventually leads to almost linear scaling of the whole application (Chart 4).

9.2.3 Multiple cores

The drawback of the memory optimization described in this paragraph is that it addresses memory scaling problem only for two cores. The author was unable to generalize the method for more cores. To use more than two cores, the set of cores needs to be split into two groups. One group of cores is calculating even part, the other group is calculating the odd part. The main loop of Hopping-Matrix function can be parallelized in an OpenMP style. Some experiments needs to be performed to determine the optimal granularity of loops parallelism.

9.2.4 Results

The best results were obtained for Machine 1. The reason for this was significant dominance of memory time over CPU time on two cores (Chart 2). After implementation of the algorithm described in this paragraph the memory overhead was reduced, which resulted in a very efficient application.

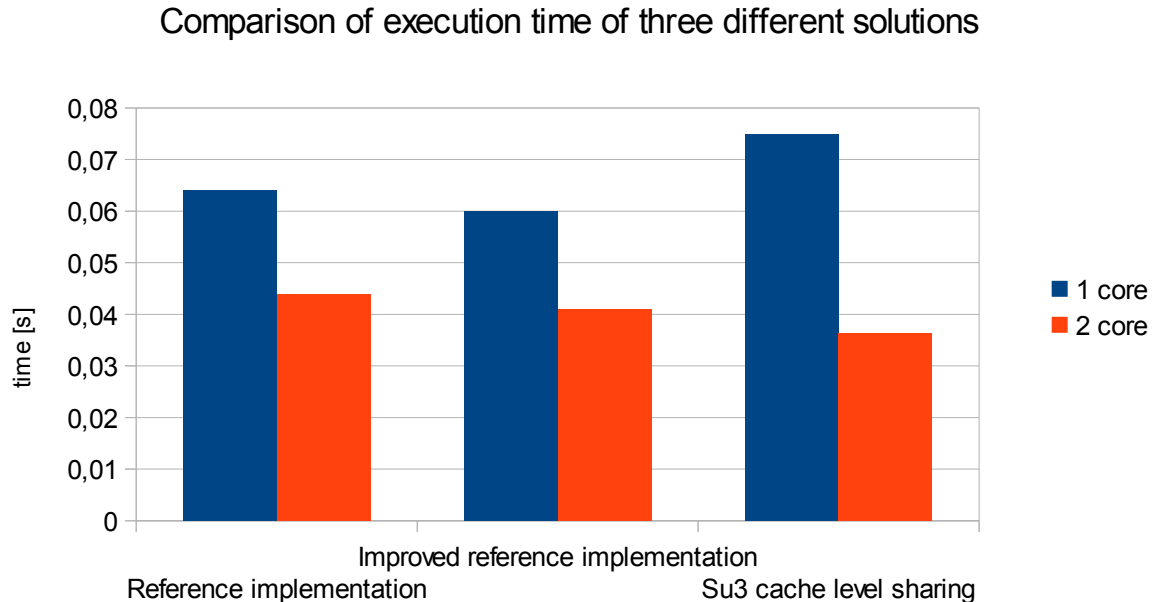


Chart 4: Comparison of results for three different implementations executed on 1 and 2 cores on the Machine 1.

The Improved reference version was 6.8% faster than the reference version. The su3 cache level sharing implementation was 17% faster than the reference version.

10 Related work

As presented in paragraph 9, the Hopping-Matrix function can be both memory and CPU bounded depending on a machine. Modifications to Hopping-Matrix function code need to be designed for a specific architecture. Michael Clark in [13] presents an algorithm which reconstructs a su3 matrix (18 double values) from 12 or just 8 double values. This solution is designed for GPUs to save costly memory accesses. Such an algorithm could also suit a multi-core, low frequency processors.

11 Tools and measures

To analyze performance of an algorithm several characteristics are collected using appropriate tools.

11.1 Cachegrind

Cachegrind is a part of the open source Valgrind software which given an application and processor description (L1, L2 cache size, cache policy) simulates the cache behavior and provides detailed information such as:

- Number of instruction reads Ir, Ir L1 cache misses, Ir L2 cache misses.
- Number of memory reads Dr, Dr L1 cache misses, Dr L2 cache misses.
- Number of memory writes Dw, Dw L1 cache misses, Dw L2 cache misses.
- Number of branch conditions Bc, Bc misses.

11.1.1 Disadvantages

The Cachegrind tool doesn't tracks the impact of software and hardware prefetches.

11.2 VTune

VTune is a commercial Intel application designed to analyze software performance. During an execution of an analyzed program processor events are registered. The data collected in the first phase is presented to the user in clear graphical way. The VTune analysis tracks far more processor events, however, the counts for each event are less precise than those of the Cachegrind.

11.3 Tools comparison

The principle of operation of vTune is different from the Cachegrind. It is not a simulator, but it records live processor performance, so it may be interrupted by other processes running (even VTune itself). Cachegrind simulates a perfect situation where only a single process is being executed on a processor. To ensure that the theoretical values obtained with Cachegrind are realistic the comparison with vTune is necessary.

12 Results

The following chapter consists of experiments descriptions and results. Each result is supplied with a brief note to explain the reasons for such score and suggest improvement. Each experiment was performed with su3 matrices double allocation, so it is not mentioned in algorithms description. The tool used to provide the algorithm details was Cachegrind. Time result provided for each experiment is the fastest execution out of twenty.

12.1 Reference version algorithm

The design of the reference version was already mentioned in the paragraph 3.

Symbol	Meaning
Ir	Number of instruction reads.
D1mr	Number of L1 data cache misses.
DLmr	Number of least level data cache misses.

Table 31: Explanation of the abbreviations used.

Machine	Machine 2
Loop	As in paragraph 4.4.1, even-odd preconditioning
Neighbor access	As in paragraph 4.3.1
Permutation	none

Memory optimization				none			
Time [s]	0.0381	Ir	171,507,751	D1mr	2,320,288	DLmr	1,470,769

12.2 The fastest so far

The improvement stems from two modifications. The first one is modification of iup and idn arrays, so that they contain indices in even-odd convention. This was computed before entering Hopping_Matrix function. The second is the zigzag2D strategy to traverse the lattice.

Machine				Machine 1			
Loop				Lexicographical, as in 4.4.1			
Neighbor access				As in paragraph 4.3.1			
Permutation				Zigzag2D			
Memory optimization				none			
Time [s]	0.0297	Ir	161,808,391	D1mr	2,276,304	DLmr	1,424,229

12.3 Best reduction of D1mr

The reduction of D1mr is significant, however, the execution time is huge. There could be multiple reasons for this issue. Firstly it could be the DLmr (data least level (L2) cache miss read) that vastly contribute to the time spent in the Hopping_Matrix function. Moreover the number of instruction reads was higher than in other experiment. Finally the D1mr is theoretical value that would be received in case when only one program is being executed on a processor, other processes executed at the test machine could have spoiled the result.

Machine				Machine 2			
Loop				As in paragraph 4.4.1, even-odd preconditioning			
Neighbor access				As in paragraph 4.3.1			
Permutation				Modified Zigzag4D (paragraph 5.1.5) so that 4×2^3 , instead of 2^4 tile is processed.			
Memory optimization				As in paragraph 6.2. Parameter $4 \cdot R = 16$.			
Time [s]	0.0389	Ir	173,015,05	D1mr	1,980,677	DLmr	1,423,852

Table of Contents

1	The Hopping_Matrix function.....	1
1.1	Dirac Operator.....	1
1.2	Data structures.....	1
1.3	Memory requirements.....	2
1.4	Optimization.....	2
1.4.1	Single core processor.....	2
1.4.2	Multi-core processor.....	3
1.5	The limit for memory access optimization.....	3
1.5.1	L1, L2 Cache misses.....	3
1.5.2	32 x 163 problem size.....	4
1.5.3	32 x 163 even-odd preconditioning.....	5
2	Hardware.....	5
2.1	Processor.....	5
2.1.1	Cache design.....	6
	Fully associative cache.....	6
	Direct-mapped cache.....	6
	N-way associative cache.....	6
2.2	Memory read latency.....	7
3	Current Hopping_Matrix implementation.....	7
3.1	Memory alignment.....	7
3.2	Tiling.....	7
3.2.1	Tiling memory requirements.....	8
3.2.2	32 x 163 problem size.....	8
3.2.3	32 x 163 problem size conclusions.....	9
3.3	su3 matrices double allocation.....	9
3.4	Other optimizations.....	10
4	Neighbor spinors access.....	10
4.1	Data structure.....	10
4.2	Even-odd preconditioning.....	10
4.3	Accessing neighbor nodes.....	11
4.3.1	Memory based access.....	11
4.3.2	Computation based access.....	11
4.3.3	Binary operations based access.....	12
4.4	Even-odd lattice traversal methods.....	12
4.4.1	Lexicographical approach.....	12
4.4.2	Naive approach.....	13
4.4.3	Improved approach.....	13
4.4.4	Singe variable loop.....	13
4.5	Experiment.....	14
4.5.1	Memory usage decrease.....	15
4.5.2	Instruction read increase.....	15
5	Permutation optimization.....	15
5.1	L1 data cache usage optimization.....	16
5.1.1	Lexicographical indexing.....	17

5.1.2	Zigzag2D numbering.....	17
5.1.3	Improved Zigzag2D numbering.....	17
5.1.4	Zigzag3D.....	18
5.1.5	Zigzag4D.....	18
5.2	Experiment.....	19
5.2.1	Conclusion.....	21
6	Cache optimization.....	21
6.1	Cache pollution.....	21
6.2	Solution.....	22
6.3	Performance.....	24
6.4	D1mr experiment.....	24
6.5	Hardware prefetch.....	25
7	Data prefetching.....	25
7.1	Hardware prefetching.....	26
7.1.1	Least level cache prefetching.....	26
7.2	Hardware prefetch impact on Hopping-Matrix.....	26
7.2.1	Prefetching spinors.....	26
7.2.2	Prefetching su3 matrix.....	27
7.2.3	Reference implementation.....	27
7.2.4	Zigzag.....	27
7.3	Software prefetch.....	27
7.4	Software prefetch impact on Hopping-Matrix.....	28
7.4.1	Su3 L2 cache pollution.....	28
7.4.2	Su3 prefetch strategy.....	28
8	Hybrid algorithms.....	28
8.1	Improved Zigzag2D split into halves.....	29
9	Tools and measures.....	30
9.1	Cachegrind.....	30
9.1.1	Disadvantages.....	30
9.2	VTune.....	30
9.3	Tools comparison.....	30
10	Results.....	31
10.1	Reference version algorithm.....	31
10.2	The fastest so far.....	31
10.3	Best reduction of D1mr.....	31

Bibliography

- 1: F. Wilczek, , 2000
- 2: C. Taroni, An efficient CELL library for Lattice Quantum Chromodynamics,
- 3: Mark D. Hill and Alan Jay Smith, Evaluating associativity in CPU caches, 1989
- 4: Dr. Colin Keng-Yan Tan, CS 1104 Caches, ,
- 5: Franck Delattre et Marc Prieur, www.behardware.com/articles/623-6/intel-core-2-duo-test.html, 2006,
- 6: , http://wwwcdf.pd.infn.it/valgrind/cg_main.html, ,
- 7: Ulrich Drepper, What Every Programmer Should Know About Memory, 2007
- 8: , Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2011
- 9: Binny S. Gill and Luis Angel D. Bathen, AMP: Adaptive Multi-stream Prefetching in a Shared Cache,
- 10: , docs.cray.com/books/S-2315-50/,
- 11: Janis Johnson, Data Prefetch Support, ,
- 12: , gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/Optimize-Options.html, ,
- 13: M. Clark, Blasting through systems of linear equations using GPUs,

ANNEXE

Internship summary.

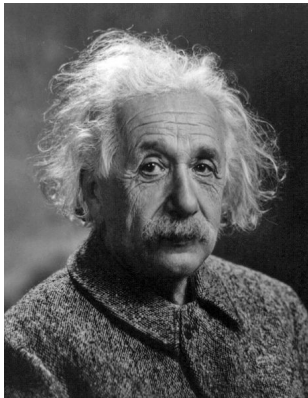
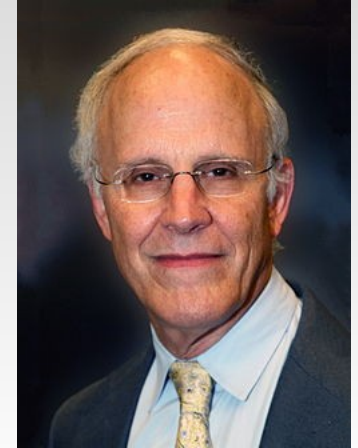
- **Who?** Wiktor Olko
- **When?** 1st of March – 30th of June 2012
- **Where?** MINES ParisTech - Fontainebleau
- **What?** High-level code optimization (case study of the Dirac Operator), for the research project PetaQCD.

Agenda

- Physics revision – LQCD.
- Current implementation.
- My contribution.

QCD

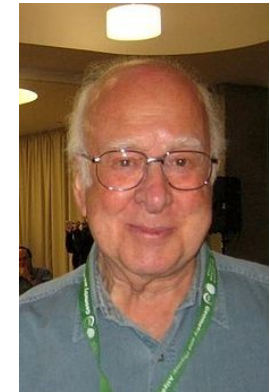
- Quantum Chromodynamics – the current theory of strong interaction (strong nuclear force).
- D. Politzer, F. Wilczek, D. Gross (photo) (early 1970s)



General Relativity
(Gravitation)
Albert Einstein (photo)
(1915)



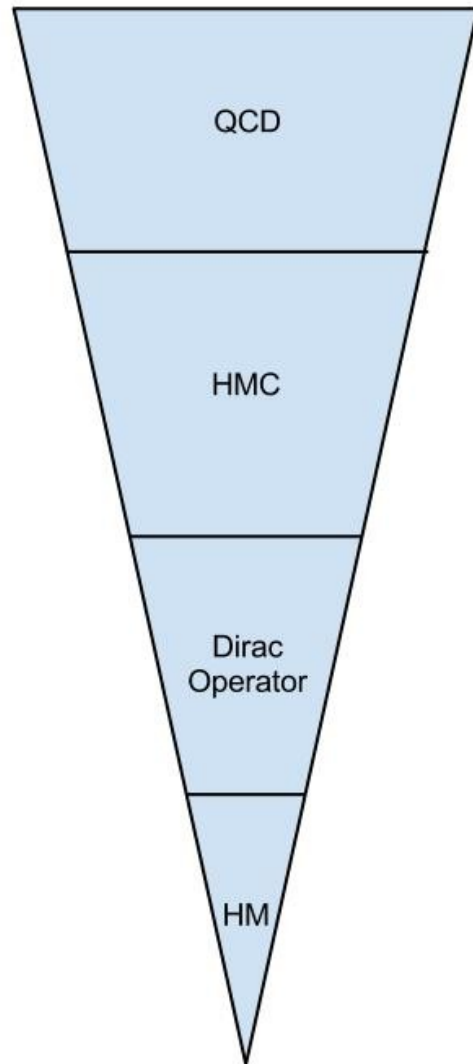
Quantum electrodynamics
(Electromagnetism)
P. Dirac (1927), R. Feynman
(photo), F. Dyson, J. Schwinger,
S. Tomonaga (1940s).



Electroweak Theory
(Weak interaction)
S. Glashow, A. Salam, S.
Weinberg (1968)
(photo: P. Higgs).

LQCD

- Numerical QCD studies are performed using discrete formalism formulated on a four dimensional grid (space and time)
- About 80% of time in the Hybrid Monte-Carlo simulations is spent to compute inversion of Dirac Operator.
- The inversion is performed using iterative methods such as: CR or GCR (Generalized Conjugate Residual).
- Hopping_Matrix function computes Dirac Operator.



Dirac Operator

$$D\psi(X) = A\psi(X) - \frac{1}{2} \sum_{\mu=0}^3 \{ [(I_4 - \gamma_\mu) \times U_{X,\mu}] \psi(X + \hat{\mu}) + [(I_4 - \gamma_\mu) \times U_{X-\hat{\mu},\mu}] \psi(X - \hat{\mu}) \}$$

Formula symbol	Meaning
$U_{x,\mu}$	3x3 complex matrix (su3 matrix)
A	12 x 12 complex matrix
$\psi(x)$	12 components complex vector (<u>spinor</u>)
I_4	Identity matrix of order 4
γ_μ	4 x 4 Dirac gamma matrices
μ	μ^{th} vector of canonical basis i.e. $\hat{0} = (0,0,0,1)$, ..., $\hat{3} = (1,0,0,0)$

The task

- Optimize the Hopping_Matrix function
- Input: 32×16^3 lattice.
 - Each node: 192B (spinor)
 - Each vertex: 144B (su3)
- Output: 32×16^3 lattice.
 - Each node: 192B (spinor)
 - [Vertices remain unchanged]

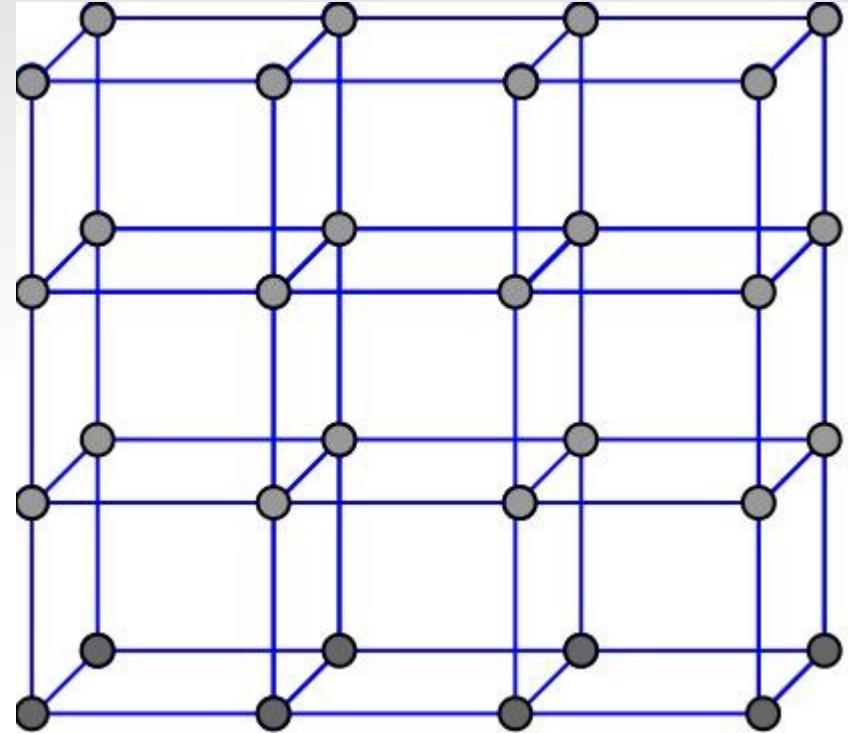
Memory requirements

Lattice size	Memory usage [GB]
$32 \cdot 16^3$	0.1 GB
$64 \cdot 32^3$	2.0 GB
$128 \cdot 64^3$	32.2 GB
$256 \cdot 128^3$	515.3 GB

Lattice

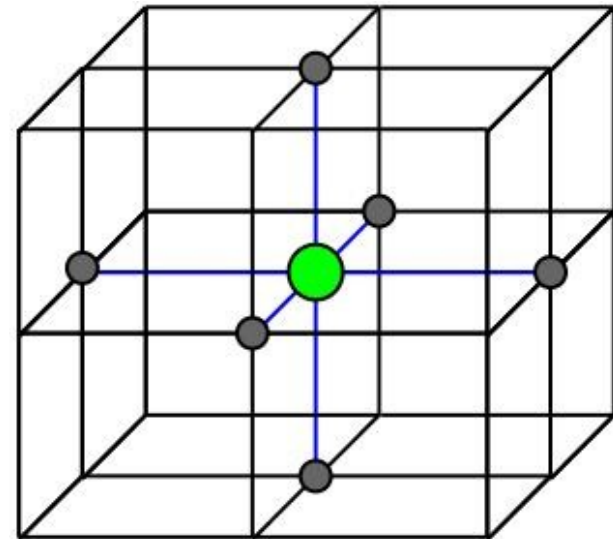
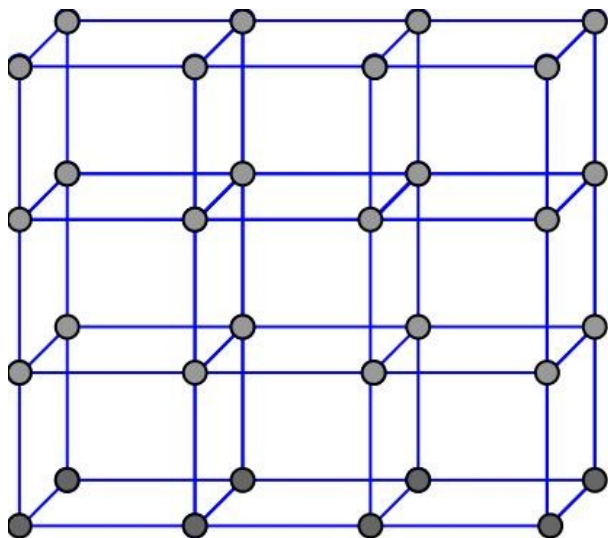
Since it is difficult to visualise 4D lattice, throughout the presentation a 3D lattice will be used.

Node \rightarrow spinor
Edge \rightarrow su_3



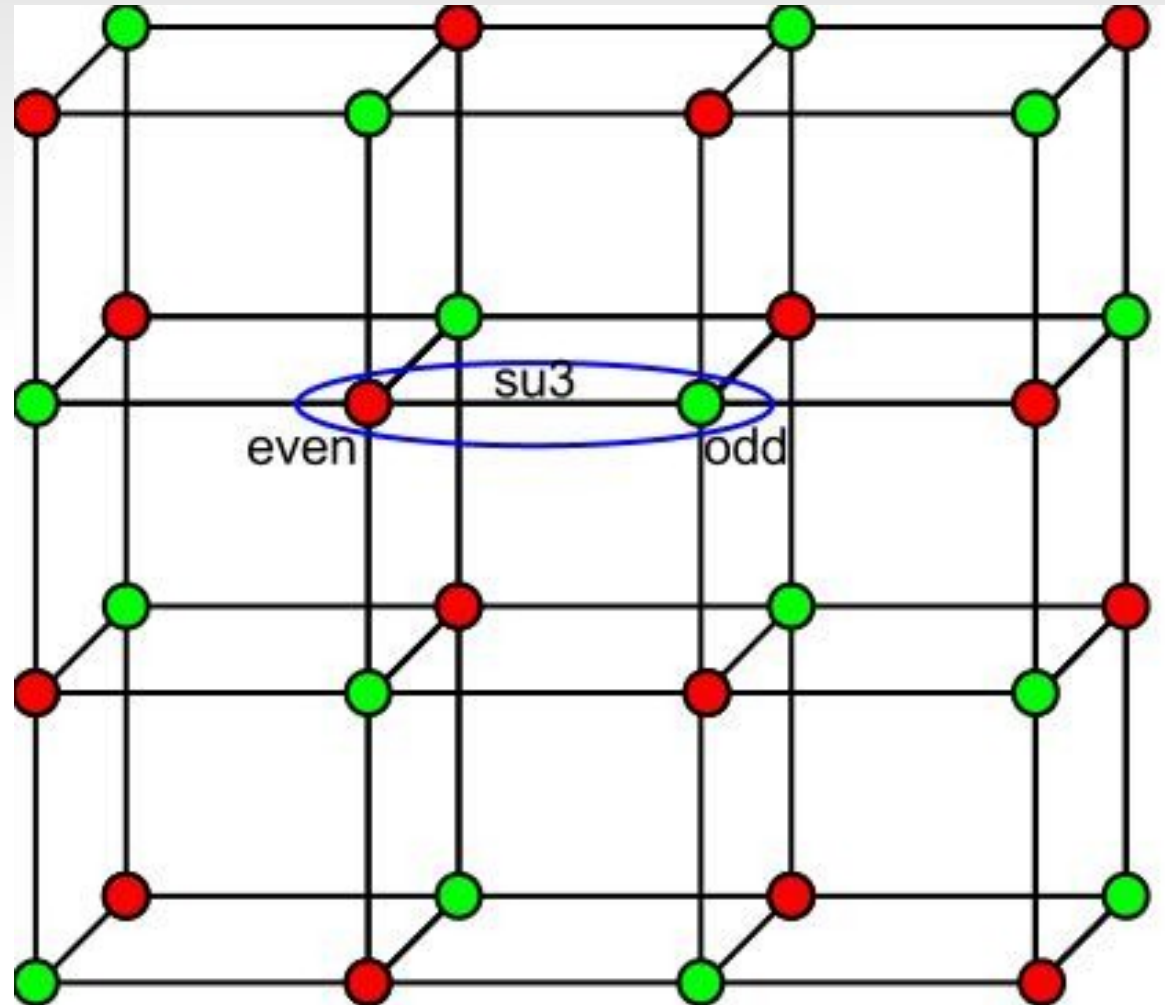
Basic implementation

- Foreach **spinor** in the lattice:
 - Foreach direction D:
 1. Read su_3
 2. Read spinor
 3. Perform computation



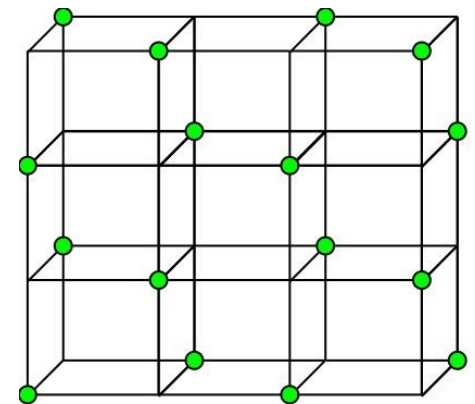
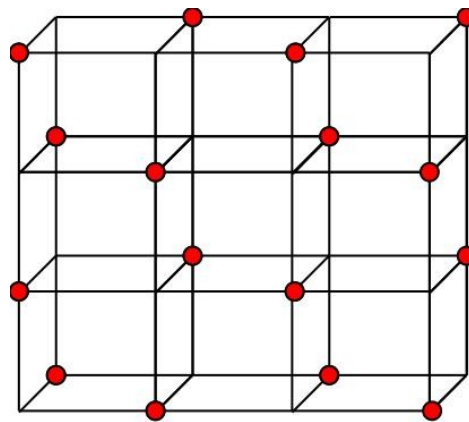
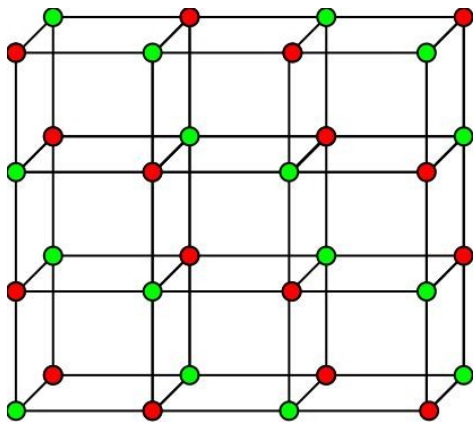
Even-odd preconditioning

- Even nodes marked in red.
- Odd nodes marked in green.
- Each pair (even,odd) shares a su_3 matrix. Marked in blue.



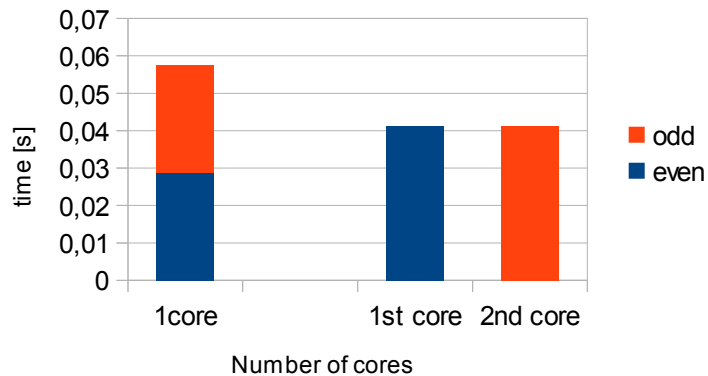
Consequences of even-odd

- Su3 double allocation, trade-off:
 - Double the su3 memory use.
 - For each spinor, all 8 neighbor su3 are stored in a continuous memory space → fast access.
- Parallelism.
- Locality.



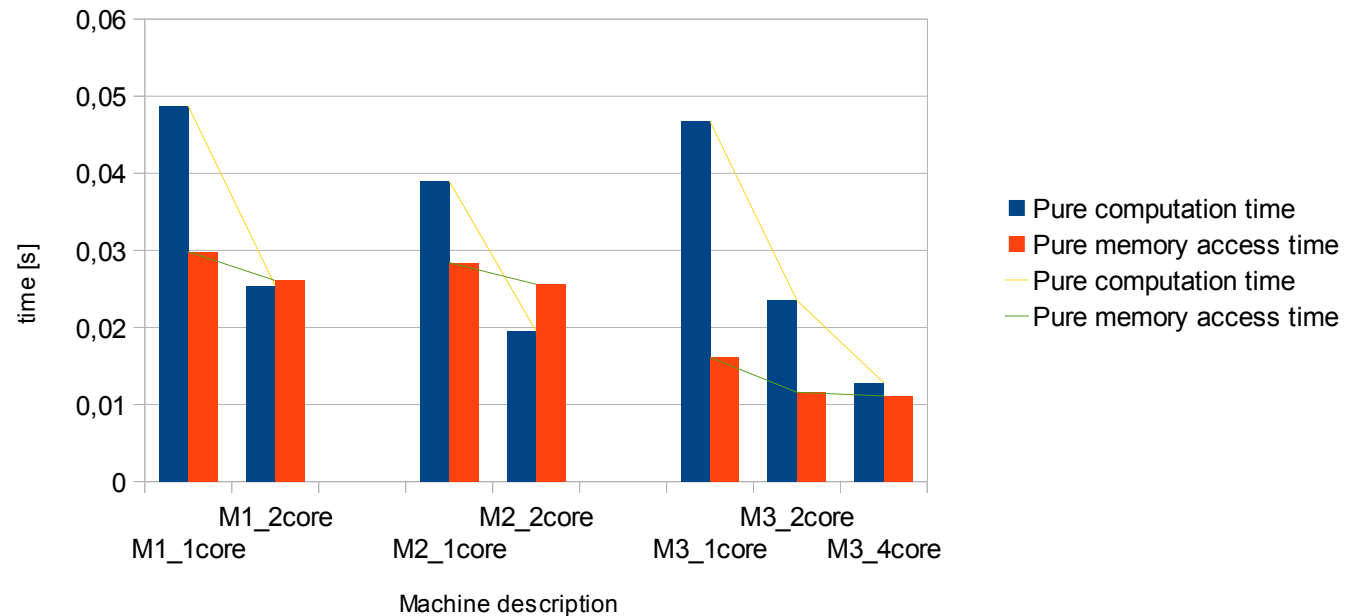
Memory limits

Scaling of the even-odd preconditioning



Limited memory bandwidth makes it impossible to load twice as much memory for two cores in the same time as for a one core.

Scaling of the CPU and the memory on multiple cores

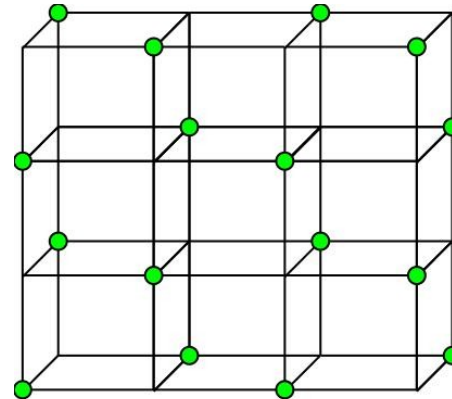
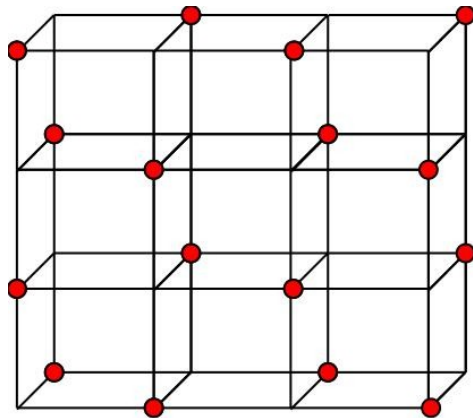


To identify if Hopping_Matrix function is CPU or memory bounded two applications have been created:

1. Pure computations (all memory accesses were removed).
2. Pure memory accesses (all computations were removed).

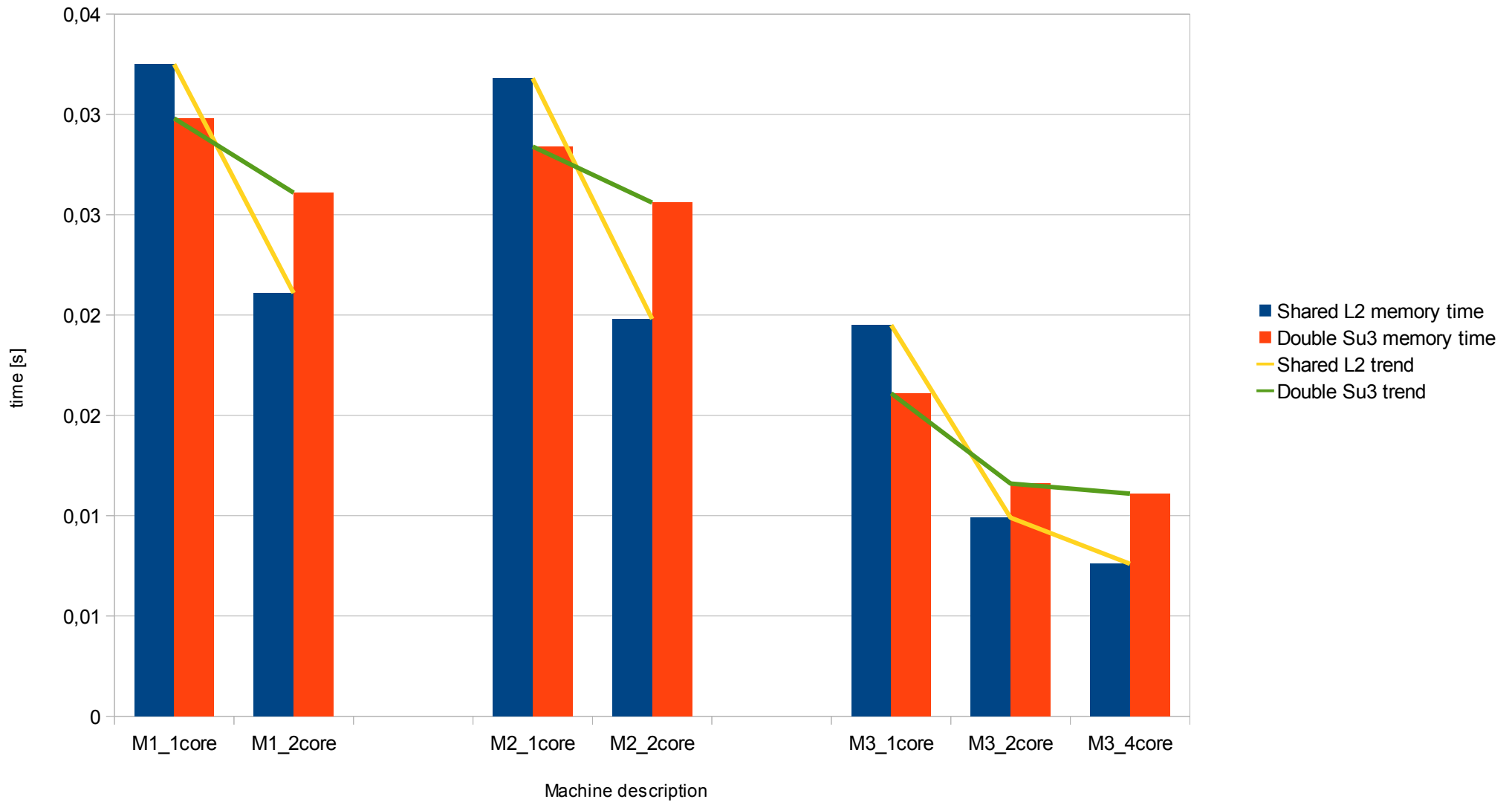
My contribution

- Perform calculations on the even and odd lattice in parallel to reuse the common su_3 matrix at L2/L3 cache level.
- Resign from su_3 double allocation



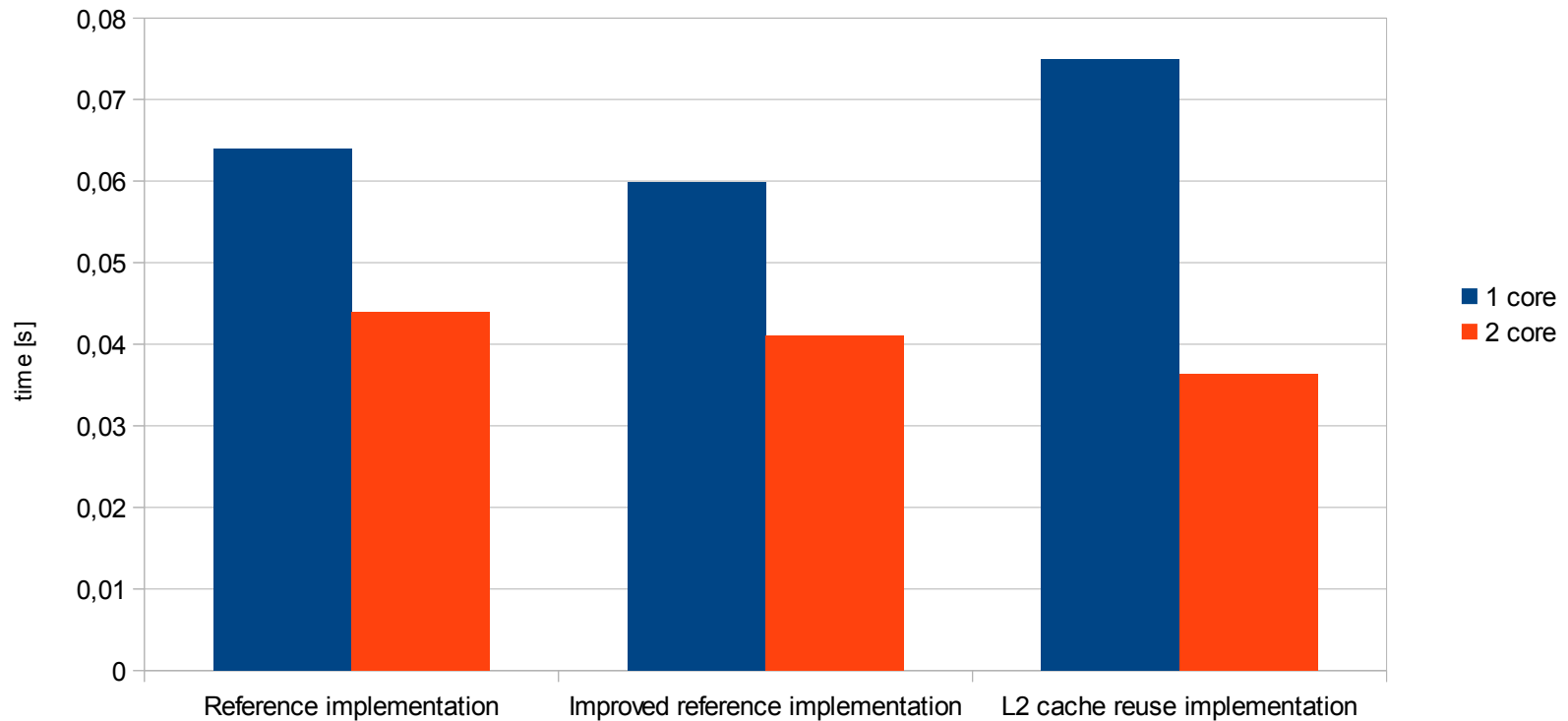
Results

Comparison of memory scaling on different machines.



Efficiency

Comparison of execution time of three different solutions



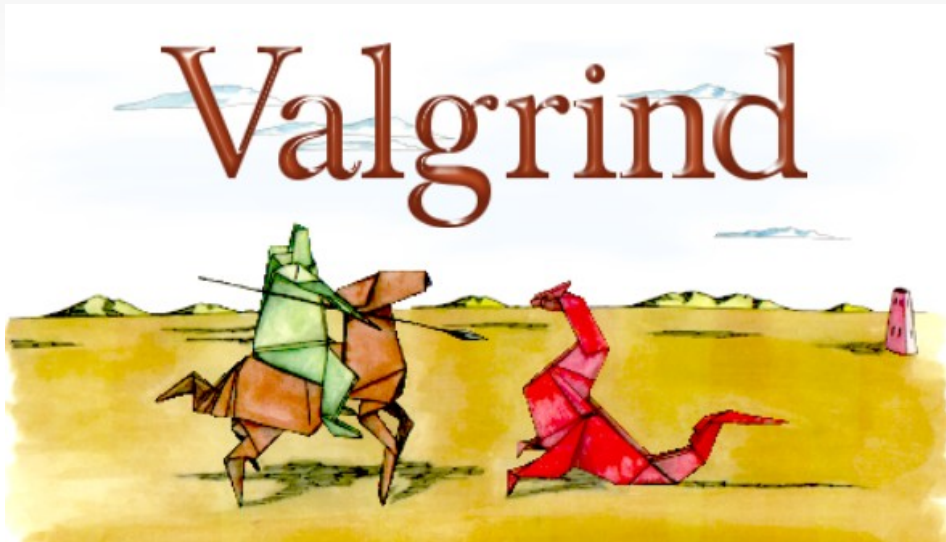
	Reference implementation	Improved reference implementation	L2 cache reuse implementation
Efficiency	0.72	0.73	1.03

Improvements

Comparison with the reference implementation.

	Improved reference implementation	L2 cache reuse implementation
Relative time execution improvement (2 cores)	6.8%	17.3%
Reasons for improvement	<ul style="list-style-type: none">• Two accesses <code>g_lexic2eosub[idn[i]]</code> replaced with only one.• 8^4 tile size.• Altered order of traversing a tile to improve L1 cache reuse.	<ul style="list-style-type: none">• Two accesses <code>g_lexic2eosub[idn[i]]</code> replaced with only one.• 16×8^3 tile size.• The new approach to memory loading.

Profiling tools



Agenda

- Valgrind, VTune:
 - How does it work?
 - What information does it provide?
 - How to use it?
- Comparison of the tools.

What is Valgrind?

- GNU General Public License, version 2.
- Instrumentation framework for building dynamic analysis tools.
- Set of tools:
 - Memcheck, SGcheck (memory errors)
 - Cachegrind, Callgrind (cache profilers)
 - Helgrind, Drd (thread errors)
 - Massif, DHAT (heap profiler)
- Support: X86/Linux, AMD64/Linux, ARM/Linux, PPC32/Linux, PPC64/Linux, S390X/Linux, ARM/Android (2.3.x), X86/Darwin and AMD64/Darwin (Mac OS X 10.6 and 10.7)

Valgring operating principle

- No need to: recompile, relink or modify application under test (-g compile option recommended).
- Executable is run on a virtual CPU provided by the Valgrind core.
- The selected tool adds its own instrumentation code, then the app is executed on Valgrind core.
- The selected tool collects information from Valgrind core and presents the output (text file).

Memcheck: a memory error detector

- Default tool, no need to specify it for Valgrind.
- Common memory problems in C and C++:
 - Accessing memory you shouldn't (heap or stack)
 - Using undefined values
 - Incorrect freeing of heap memory
 - Memory leaks.



Memcheck: sample output

Sample application

```
int f(){
    int *a=(int*)malloc(8);
    return 4;
}

int main(){
    int *a = (int*)malloc(8);
    int c = f()+a[5];
    free(a);
}
```

Sample output

```
Invalid read of size 4
    at 0x8048455: main (v.c:13)
Address 0x41c403c is 12 bytes
    after a block of size 8 alloc'd
```

HEAP SUMMARY:

```
in use at exit: 8 bytes in 1
    blocks
total heap usage: 2 allocs, 1
    frees, 16 bytes allocated
```

LEAK SUMMARY:

```
definitely lost: 8 bytes in 1
    blocks
```

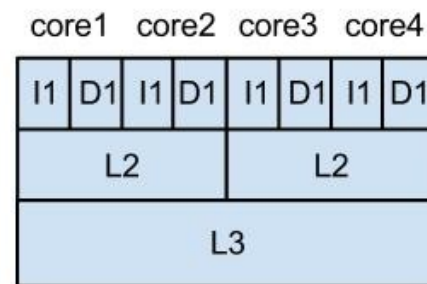
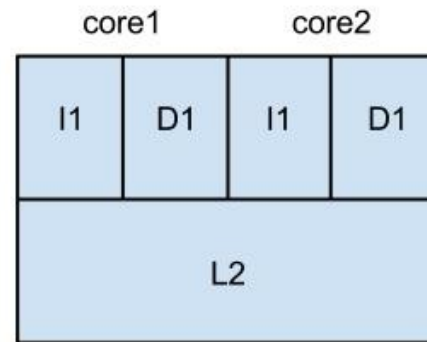
Cachegrind: a cache and branch-prediction profiler

- Simulates how your program interacts with a machine's cache hierarchy and branch predictor.
- Unified cache hierarchy.

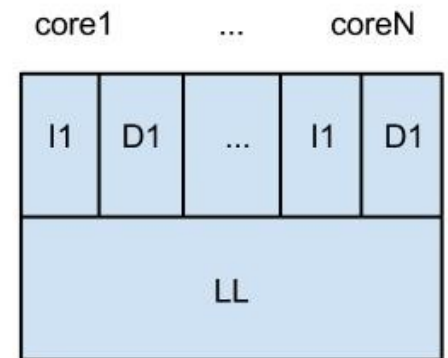
- Collected events:

- Ir
- Dr, D1mr, DLmr
- Dw, D1mw, DLmw
- Bc, Bcm

Example processors



Model processor



Cachegrind: sample output

I1 cache: 32768 B, 64 B, 8-way associative
D1 cache: 32768 B, 64 B, 8-way associative
LL cache: 6291456 B, 64 B, 24-way associative
...

Ir	Dr	D1mr	DLmr	function
----	----	------	------	----------

344,719,420	142,868,498	4,797,681	2,996,387	Hopping_Matrix
84,488,688	12,782,562	393,234	393,234	main
39,321,600	10,485,760	1,572,865	1,572,865	__intel_sse3_me

What is VTune?

- Commercial Intel product – standalone or included in a suite (Parallel Studio, C++ Studio)
- Facilitates the full process of improving software and the way it uses available resources.
- Two analysis categories: Algorithm Level and Advanced Hardware-level Analysis.

	Windows	Linux
GUI	Visual Studio integrated	Standalone client
Command line interface	yes	yes

VTune - Algorithm Analysis

- User-Mode Sampling and Tracing Collection
 - Snapshot of the number of threads at a given moment provide a hint to the degree of parallelism.
 - Sampling interval: 10 ms for Linux* 2.4; 1,2 or 4 milliseconds for new Linux ≥ 2.6
 - Use collected data to understand the control flow for statistically important code sections.
- General information about hotspots in the application.

VTune - Advanced Hardware-level Analysis

- Hardware Event-based Sampling Collection
 - Uses installed driver to configure and collect interrupts from the Performance Monitoring Unit of each Intel CPU Core.
 - PMU has limited number of simultaneous events it can track (max 3 or 4).
 - Example events: L1/L2 cache misses, clock cycles, instructions retired, hardware/software prefetches.
 - User specifies: events to track, "sample after" value.
- Very detailed information.

VTune GUI

The screenshot displays the Intel VTune Amplifier XE 2011 interface. The main window shows a table of hardware event counts for the analysis target 'L2 cache both cores'. The table includes columns for source addresses, assembly instructions, and various hardware events. The event 'MEM_LOAD_RETIRED.L2_LINE_MISS' is highlighted in orange, with a value of 300. Other events include 'MEM_LOAD_RETIRED.L2_MISS' (60,000), 'L2_ADS.BO...' (489,000), and 'L2_DBUS...' (815).

Project Navigator: /home/olko/intel/amplxe/projects

ImprovedNaive

- r006l2-custom-1
- r007l2-custom-1
- r008l2-custom-1
- JuneLexic
- MemBased

r006l2-custom-1

L2 cache both cores - Hardware Event Counts

Analysis Target | Analysis Type | Summary | Bottom-up | alone

Source | Assembly | MEM_LOAD_RETIRED.L2_LINE_MISS | MEM_LOAD_RETIRED.L2_MISS | L2_ADS.BO... | L2_DBUS...

Source	Assembly	MEM_LOAD_RETIRED.L2_LINE_MISS	MEM_LOAD_RETIRED.L2_MISS	L2_ADS.BO...	L2_DBUS...
0x8049b63	shl \$0x6, %edx				1,
0x8049b66	lea (%esi,%esi,8), %e			1,000	
0x8049b69	shl \$0x6, %esi				
0x8049b6c	movl %esi, 0x5d8(%es		100	1,000	7,
0x8049b73	movsdq (%edi,%ecx,1				
0x8049b78	movsdq 0x8(%edi,%ec	25,300	60,000	489,000	815,
0x8049b7e	movsdq (%esi,%ebx,1	300	900	17,000	28,
0x8049b83	movaps %xmm5, %xmm7	7,600	15,000	120,000	183,
0x8049b86	addsdq 0x60(%edi,%e			1,000	1,
0x8049b8c	addsdq 0x68(%edi,%e	10,700	28,500	150,000	235,
0x8049b92	mulsd %xmm6, %xmm7	5,800	400	9,000	12,
0x8049b96	mulsd %xmm0, %xmm5		200	18,000	22,
0x8049b9a	movsdq %xmm6, 0x3e8			19,000	17,
0x8049ba3	movsdq 0x8(%esi,%eb				
0x8049ba9	movsdq 0x10(%edi,%e		1,000		
0x8049baf	mulsd %xmm0, %xmm6	100	9,000	3,000	9,
0x8049bb3	addsdq 0x70(%edi,%e			8,000	9,
0x8049bb9	movsdq 0x18(%edi,%e	600	18,100	1,000	11,
0x8049bbf	subsd %xmm6, %xmm7		5,700		8,
0x8049bc3	addsdq 0x78(%edi,%e	100		44,000	29,
0x8049bc9	movsdq 0x10(%esi,%e	100	2,800	5,000	14,
Selected 1 row(s):		300			

No filters are applied.

Timeline Hardware Event: MEM_LOAD_RETIRED.L | Inline Mode: on

Valgrind vs VTune

	Valgrind	VTune
Number of processes analysed	One	Every process running.
Operating principle	Core model, dynamical application instrumentation.	Hardware PMU events / User-Mode Sampling
Licence	GNU GPL	Commercial (free: home use, trials, beta releases; discount for: students, universities).
Source	Open, possible to modify when needed, contribute.	Closed.

Sources

- <http://valgrind.org/docs>
- http://www.rz.rwth-aachen.de/global/show_document.asp?id=aaaaaaaaaccile
- Intel(R) VTune(TM) Amplifier XE 2011 docs.