Efficient linear algebra related to lattice QCD on Cell Broadband Engines

Martin Wolf
Ruprecht-Karls-University Heidelberg

mail@martin-wolf.org

Aim of the project was to investigate several possible implementations of a SU(3) matrix on the Cell Broadband Engine (CBE) processor according to their efficiency. The motivation for such an investigation comes from lattice QCD, where efficient linear algebra with SU(3) matrices is fundamental to be able to handle the huge amount of calculations in a finite time. With faster algebra software libraries on parallel supercomputers it could be possible to increase the used lattice volume and to decrease the lattice spacing a to obtain more precise results for quark-gluon and gluon-gluon interactions. The C++ software library, I wrote, contains two possible implementations for a SU(3) matrix. It hides the complexity of the processor architecture. So programmers are able to write lattice QCD applications more easily and independent from the used architecture. The library's efficiency was investigated by myself with benchmark tests which have been executed on a QSZO blade with two Cell Broadband Engine processors each located at the research center in Jülich, Germany.

Contents

1 Introduction

At first a small motivation is given why efficient linear algebra is needed from the physics point of view. Than an explanation of the used development environment and an overview of the Cell Broadband Engine follows, as well as an introduction on how to measure time on a Cell processor. The document will continue then with the description and usage of the newly written C++ library for the CBE and finally the benchmark results are presented and discussed.

1.1 Motivation - Why efficient linear algebra?

The final aim is to simulate elementary particles interacting to each other by the strong force. Strong interacting particles can be described by quantum chromodynamics (QCD) what is a gauge theory based on the non-abelian SU(3)-colour group. To simulate this theory on the computer one has to discretize the space-time onto a lattice of size V and a lattice spacing a. The quark fields are than only defined at the nodes of the lattice. Instead of a vector potential as in the continuum case, the gauge fields variables are defined on the links of the lattice and correspond to the parallel transport along the edge which takes on values in the Lie group. Hence to simulate QCD, for which the Lie group is SU(3), there is a 3 by 3 special unitary matrix defined on each link. The faces of the lattice are called plaquettes. To calculate a quantity (such as the mass of a particle) in lattice gauge theory, it should be calculated for every possible value of the gauge field on each link, and then averaged. In practice this is impossible. That's why the Monte Carlo method is used to estimate the quantity. An importance sampling is done and these chosen configurations (values of the gauge fields) are generated with probabilities proportional to $e^{-\beta S}$, where S is the lattice action for that configuration and β is related to the lattice spacing a. Observables of interest are calculated on each configuration. Estimates for these observables are then found by taking the average from a large number of configurations. To find the value of an observable in the continuum this procedure has to be repeated for various values of a and then an extrapolation to $a = 0$ has to be performed. All algorithms for simulating QCD on a lattice involve a fair amount of linear algebra operations. For this exploratory study we restricted ourselves on operations with SU(3) matrices. Our approach can, however, easily be applied to linear algebra operations involving other types of complex objects. For further reading on lattice QCD I refer to a book from Heinz J. Rothe [Rot97].

1.2 Setup – The used development environment

The here described software, the C++ library and the benchmark tests, have been compiled using the IBM Software Development Kit (SDK) version 2.0. This software kit contains the essential tools required for developing programmes for the Cell Broadband Engine. I used the GNU C++ compiler "gcc" version 4.1.1, the IBM Full System Simulator for the Cell Broadband Engine (`systemsim`) and a static timing analysis timing tool, `spu_timing`, that instruments assembly source with expected instruction timing details.

1.3 Overview of the Cell Broadband Engine Processor

A detailed description of the CBE Processor is given in the IBM documentation [IBM06]. Here I will just give a brief general introduction on the processor architecture. Figure 1 shows an overview of the processor. The CBE consists of one main multi-threaded processor called Power Processor Element (PPE) where the operating system is running on as well as the application programme. Furthermore there are eight sub-processors called Synergistic Processor Elements (SPEs) which are actually the workhorses doing the calculations in threads of the task. All nine processors are connected to each other by the Element Interconnect Bus (EIB) providing the communication between the processors. The PPE and all SPEs share the same memory through the Memory Interface Controller which is also attached to the EIB. But the PPE and the SPEs have different memory access procedures. The PPE fetches data through a cache from the main memory and stores it into its processor registers. The SPEs load data with asynchronous Direct Memory Access (DMA) transfers from the main memory and store it into their 256KByte Local Store (LS). From there the SPE loads the data into its large 128 x 128Bit register file. With this three level memory architecture the SPEs are independent from the PPE and intensive calculations won't be interrupted by DMA transfers. The CBE can reach a maximal peak performance of 8 Flops $\cdot$ 8 SPEs $\cdot$ 3.2 GHz = 204.8 GFlops/s. Because of this high peak performance CBE processors could be good candidates for supercomputing architectures in the future. You can find CBE processors as well in your everyday life, e.g. in the gaming console Sony PlayStation III.

1.4 SIMD on the SPEs

SIMD stands for Single Instruction Multiple Data and is a very important technique for parallelizing calculations on a computer. The main goal is to have one instruction and execute this once on multiple data simultaneously. For this behavior the data has to be arranged in vector data types. Figure 2 shows a typical SIMD add operation. The SPE has four floating point number pipelines, e.g. four add operations can be executed simultaneously.

Another great property of the SPE is that it has two instruction pipelines: the even (0) and the odd (1) pipeline. Each instruction belongs to one of these two pipelines which are connected to the several execution units, e.g. the FPU. This means an instruction is either an even instruction or an odd instruction. The SPE fetches its instructions pairwise in a so called fetch group and if the first instruction in the fetch group is an even instruction and the second one an odd instruction then the SPE can issue both instructions at once, e.g. the SPE can handle, execute and complete up to two instructions per cycle. But for dual-issue behavior the instructions must appear in the appropriate order and an even instruction must be followed by an odd instruction.

1.5 Time measurement on the SPEs

Always important in my project was the question: "How fast are several parts of the library?" To answer this question the time needed to execute a certain part of the programme had to be measured. But how can one measure the execution time of a calculation? Each SPE provides a so called decrementer register. This register is a 32Bit integer which decrements on a constant rate based on the SPE clock frequency. Setting this number to a high value and reading it two times – once when the calculation starts and once when it finished – you get a value indicating the elapsed time.

2 The newly written Cell C++ library

The C++ library I wrote consists of several C++ classes. General classes are `spu_decrementer` and `spu_mem` described in the following sections. Classes for data types like complex numbers or SU(3) matrices are `VComplex2`, `VComplex4`, `SU3_VC2` and `SU3_VC4` also described here later on.

2.1 The `spu_decrementer` class

This class offers the general functionality of SPE time measurements by using the SPE's decrementer register. One can start a time measurement by creating an object instance of this class and calling the (inline) class method `start()`. After finishing some calculations one can stop the time measurement by calling the class method `stop()`. The `start` method sets the decrementer register to a high value. Both methods, `start` and `stop`, return the current decrementer value. After executing the `stop()` method the static object member `counts` holds the elapsed decrementer count ticks. This variable is not reset by the `start` method. So one can interrupt time measurements by calling the `stop` method and can continue them by calling the `start` method a second time. To reset the counter variable one has to call the `reset` method.

2.2 The `spu_mem` class

The `spu_mem` class consists of several static member methods for checking SPU memory / DMA conditions. For example the so called *context*, a memory block transferred from the PPE to the SPE, must be a multiple of 16 bytes and less than 16 kbytes. The static method `check_dma` checks those requirements.

2.3 The `VComplex2` class

Essential for lattice QCD calculations are complex numbers. The fact that the SPE arranges its data into vectors with four single precision floating point numbers (SIMD technique), has the consequence that storing single complex numbers in each data vector is simply not efficient because one would waste half the memory. Only the storage of a multiple of two complex numbers would be memory efficient. The `VComplex2` class provides a vector of two complex numbers generated by four single precision floating point numbers (`vector float`). Arithmetic complex operations like addition, subtraction, multiplication, complex conjugation and calculating the norm of a complex number have been implemented.

2.4 The `VComplex4` class

Very similar to the `VComplex2` class is the `VComplex4` class. It provides a data type for complex numbers as well, but with the difference of holding four instead of only two complex numbers. It is implemented with two four-floating-point-number vectors. The first one holds the real parts and the second one holds the imaginary parts of all four complex numbers. This arrangement of the data has the advantage that no shuffle operations are required when multiplying two `VComplex4` objects.

2.5 The SU(3) matrix classes

Common objects in QCD are SU(3) matrices. Therefore, a SU(3) matrix data type has to be supported by the library. There are two possibilities to implement such a data type either with the `VComplex2` or with the `VComplex4` class. I implemented both. Each one has advantages and disadvantages as well. Common operations with a SU(3) matrix are the multiplication with an other SU(3) matrix and the multiplication with its adjoint (transposed and complex conjugated) matrix: $A \cdot A$, $A \cdot A^{\dagger}$. Therefore these operations have to be optimized. The table shows the required operation counts for multiplications of each implementation. With dual-issues the SPE should be able to handle an even and an odd instruction simultaneously. The maximum number of even or odd instructions, max(even, odd), therefore gives us a lower bound for the number of cycles needed to execute this particular operation.

2.5.1 The `SU3_VC2` matrix class

The first implementation of a SU(3) matrix class was done by combining five `VComplex2` class objects to one matrix object class called `SU3_VC2`. The advantage using the `VComplex2` class is that one wastes only two floating point numbers / one complex number ($2 \cdot 32$Bits $= 64$Bits) of memory per matrix because you have ten – five times two – complex numbers per matrix and a SU(3) matrix only consists of nine complex numbers. On the other hand the `VComplex2` multiplication operations need a lot of shuffle operations which is a real disadvantage. To multiply two `SU3_VC2` matrices one needs at least 89 operations and at least 48 clock cycles.

2.5.2 The `SU3_VC4` matrix class

The `SU3_VC4` matrix class is the second implementation of a SU(3) matrix which uses three `VComplex4` objects for holding all nine complex numbers. But with this storage kind one loses 25% of memory per matrix (six floating point numbers / three complex numbers ($6 \cdot 32$Bits $=$ 192Bits)). This means that on the one hand we have to execute less operations on the SPE, but we will spend more time on loading the data. To multiply two `SU3_VC4` matrices one needs at least 78 operations and at least 48 clock cycles.

3 Benchmark tests on the SPE

Benchmarking is an important topic for numerical calculations. The aim is always to minimize the execution time of a specific calculation programme. When a programmer uses a software library he expects that the used library is already optimized. That's why I had to benchmark the library routines. In the following sections I will describe my benchmark tests.

3.1 Fundamental programming language benchmarking

For programmers writing supercomputing applications it is essential to know how much execution time fundamental language constructs e.g. conditions or loops will require or in colloquial words: How much they will cost. The problem is that every condition or loop results in one or more branches within the processor's instruction queue and every branch may be really expensive and you can lose much time and efficiency just for jumping within the processor's instruction queue when you write source code with many conditions and loops. To predict the entire execution time of your program you need to know this so called branching overhead. For this reason I tried to estimate the time for a single branch of a loop. Listing 1 shows the assembler code of a single empty loop with register $2 as the loop variable.

Listing 1: Assembler code for an empty loop

```
   hbra     .L32,.L16
 .L16:
   ai       $2,$2,-1
 .L32:
   brnz     $2,.L16
```

To estimate the time one loop branch needs I put several no-operation commands (NOPs) into the loop and measured the time by using the decrementer. Figure 3 shows the results. The reason of having measured a step function here is that the SPE fetches instructions pairwise and the last NOP is dual-issued with the branch command when there is an even number of NOPs inside the loop. We fit our data using the χ^2 method to the ansatz

$$t_{\text{decr}} = N_{\text{loop}} \left[\alpha_0 + \alpha_1 \left(N_{\text{NOP}} - \text{mod}(N_{\text{NOP}} - 1, 2)\right)\right] \tag{1}$$

where α_0 is the required decrementer count for one loop branch and α_1 is the required decrementer count for one NOP instruction. The measurements were done for $N_{\text{loop}} = 10^6..10^7$ in steps of 10^6 and $N_{\text{NOP}} = 1..12$ in steps of 1 on a single SPE. The fit returned the following results:

$$\begin{aligned} \chi^2 &= 10.4485 \\ \text{ndof} &= 119 \\ \alpha_0 &= 0.00894925(3) \\ \alpha_1 &= 0.004474600(43) \end{aligned}$$

From α_2 one could get the cycles/decrementer ratio:

$$1 \text{ dcount} = \frac{1}{\alpha_1} \text{ cycles} = 223.5 \text{ cycles}$$

This means for α_0, the required time for one loop, in units of cycles:

$$\alpha_0 = t_{\text{loop}} = 2 \text{ cycles}.$$

But keep in mind that those two-cycles-branches can only be performed if the processor got a correct branch hint before the branch is executed. Such a branch hint is shown in listing 1 in line 1.

3.2 SU(3) matrix multiplication benchmarking

As I mentioned above I implemented two SU(3) matrices `SU3_VC2` and `SU3_VC4`. In the benchmark test I multiplied two arrays of SU(3) matrices with a length of 50 to 800 matrices and measured the required time for this operation:

$$u_i^{ab} \leftarrow \sum_c v_i^{ac} w_i^{cb}. \qquad (2)$$

The arrays were stored in the SPE's local store (LS). The listing 2 shows the code.

Listing 2: C++ loop with SU(3) matrix array multiplication (not unrolled)

```
SU3 *ar_su3_a, *ar_su3_b;
ar_su3_a = new SU3[N];
ar_su3_b = new SU3[N];

for(i=0; i<N; i+=1)
{
    ar_su3_a[i] *= ar_su3_b[i];
}
```

The results of all SU(3) benchmark tests are shown in figure 4. The red and green points are the results for the loop shown in listing 2. The red benchmark is the result for non-Dual-Issue behavior and the green one shows the result for Dual-Issue behavior[1]. The improvement of the execution time with dual-issued code is obvious. Now one can improve the execution time even more. This can be done with so called loop unrolling. This means that instead of only one multiplication two, three, four, etc. multiplications are done within one loop iteration. This has the effect that less loop branches occur and the compiler could be able to interleave the instructions of all multiplications in a way that dependencies between instructions (e.g. an immediate store after a load of a register) are removed and the instruction density increases. The disadvantage of this method is a longer programme code and one has to consider that the whole programme has to be transferred from the main memory to the local store (LS) of the SPE. One has to find an optimal trade-off between gaining by loop unrolling and losing by loading a larger number of instructions. An unrolled loop with an unrolling factor of 3 is shown in listing 3.

Listing 3: C++ loop: SU(3) matrix array multiplication (unrolled by 3)

```
SU3 *ar_su3_a, *ar_su3_b;
ar_su3_a = new SU3[N];
ar_su3_b = new SU3[N];

for(i=0; i<N; i+=3)
{
    ar_su3_a[i+0] *= ar_su3_b[i+0];
    ar_su3_a[i+1] *= ar_su3_b[i+1];
    ar_su3_a[i+2] *= ar_su3_b[i+2];
}
```

The figure 4 shows benchmark tests for several unrolling factors. A significant improvement is obvious until an unrolling factor of 3. From there on the instruction density is almost constant and no more instruction dependencies can be resolved. Surprisingly, the benchmark tests tell us that the SU(3) matrix implemented by the `VComplex2` class is two times faster than the one implemented by the `VComplex4` class. What is the reason for that? A look into the timed assembler code (see listing 4), which was generated by the `spu_timing` tool, shows the problem: many dependency stalls indicated by many dashes! Unfortunately, the compiler is not able to recognize these dependencies and to reorder the code. The compiler could interleave two multiplications to minimize the dependency stalls but in listing 4 one can see that it does

[1]For unknown reasons the GNU C++ compiler generates inefficient code. After compilation the even and odd instructions are always in the wrong order. To cure this inefficiency one can insert a NOP instruction to obtain the good order of the instructions.

not do so. The two multiplications remain totally separated.

Listing 4: Assembler code fragment for a SU3_VC4 multiplication

```
(1st multiplication)
1  567890                     lqd  $3,16($80)
1   678901                    lqd  $4,0($80)
1    789012                   lqd  $7,32($80)
0     ---123456               fa   $3,$3,$28
0         234567              fa   $4,$4,$24
0          -----789012        fa   $3,$3,$29
0d               890123       fa   $4,$4,$26
1d               ------345678 stqd $3,16($80)

 ...

(2nd multiplication)
1D 567890                     lqd  $3,16($19)
1   678901                    lqd  $4,96($80)
1    789012                   lqd  $7,128($80)
0     ---123456               fa   $3,$3,$28
0         234567              fa   $4,$4,$21
0          -----789012        fa   $3,$3,$29
0                890123       fa   $4,$4,$27
1                 -----345678 stqd $3,16($19)
```

4 Conclusion

The newly written C++ library provides SU(3) matrix support with two kinds of implementation on a Cell Broadband Engine architecture. The SU(3) benchmark tests show that the implementation with the `VComplex2` class is more efficient than the one with `VComplex4`. But this behavior can be explained by the inefficient code which is generated by the GNU C++ compiler. It seems that the compiler is not able to rearrange the instructions in such a way that dependency stalls are eliminated. Would it be able to do so then `VComplex4` SU(3) implementation should be faster or at least as fast as the `VComplex2` implementation.

Another benchmark allowed to investigate the execution time of a single loop branch. From my result I conclude that a loop branch costs only 2 cycles in case the branch is predicted correctly.

The developed C++ library can be used and should be extended by Cell programmers in the future to create a base software library for lattice QCD simulations on the Cell Broadband Engine processors.

5 Acknowledgments

I want to thank the whole APE group for their inexhaustible support and kindness: Dirk, Hani, Hubert, Nils and Christian. It was a real pleasure to work with them together and I had a great time. I would like especially to thank Dirk for his prolific ideas, remarks and concrete aid helping me to understand the functionality of the Cell processor and its benchmarking properly. Also, I would like to thank Karlheinz Hiller and Sabine Baer for making possible this beautiful experience as summer student in Zeuthen with the very exciting trip to DESY Hamburg.

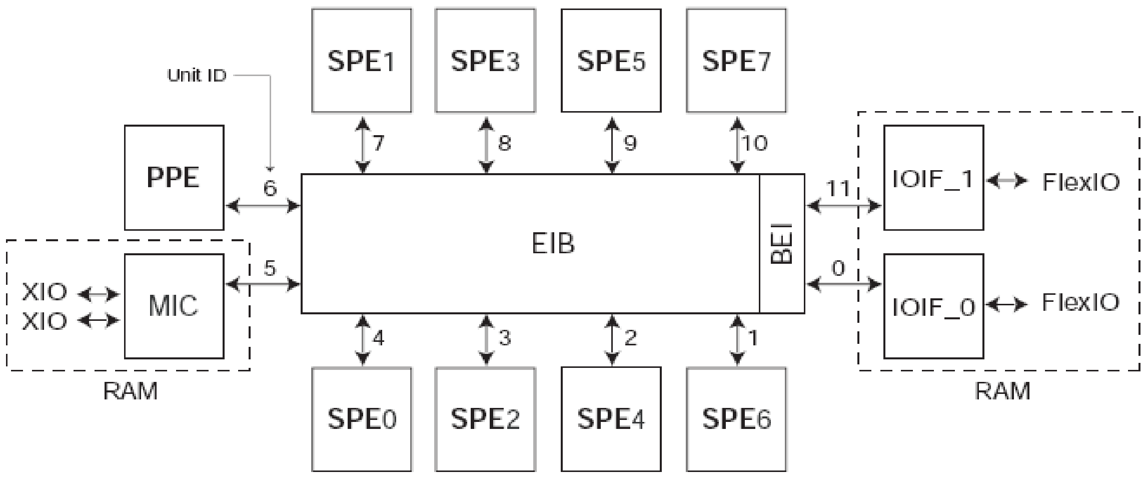
References

[IBM06] IBM. Cell broadband engine online documentation. http://www.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine, 2006.

[Rot97] Heinz J. Rothe. *Lattice Gauge Theories, An Introduction.* World Scientific Publishing Co. Pte. Ltd., P O Box 128, Farrer Road. Singapore 912805, 1997.

Table 1: Required operation counts for several multiplication operations on the SPE

instruction	`VComplex2`	`VComplex4`	`SU3_VC2`	`SU3_VC4`
float add	0	0	12	12
float multiply	2	2	18	18
float multiply add	1	1	9	9
float multiply sub	1	1	9	9
even instructions	4	4	48	48
shuffle	3	0	31	18
float load	1	2	5	6
float store	1	2	5	6
odd instructions	5	4	41	30
max(even, odd)	5	4	48	48
sum	9	8	89	78



BEI	Cell Broadband Engine Interface	MIC	Memory Interface Controller
EIB	Element Interconnect Bus	PPE	PowerPC Processor Element
FlexIO	Rambus FlexIO Bus	RAM	Resource Allocation Management
IOIF	I/O Interface	SPE	Synergistic Processor Element
		XIO	Rambus XDR I/O (XIO) cell

Figure 1: Overview of the Cell Broadband Engine Processor architecture

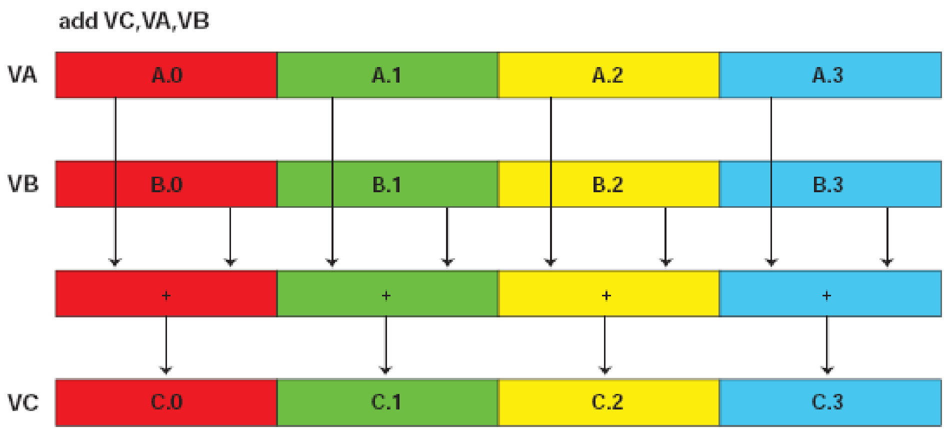


Figure 2: SIMD: Four concurrent add operations

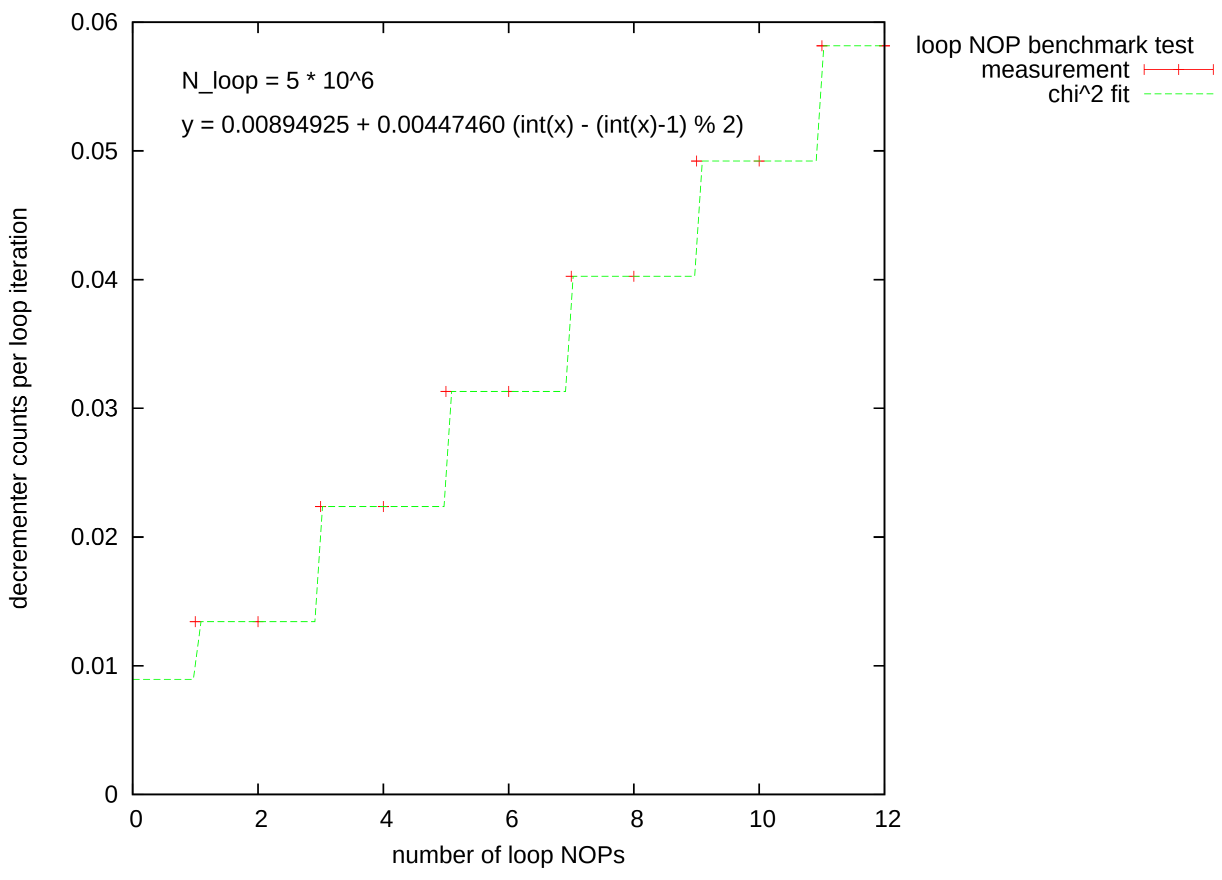
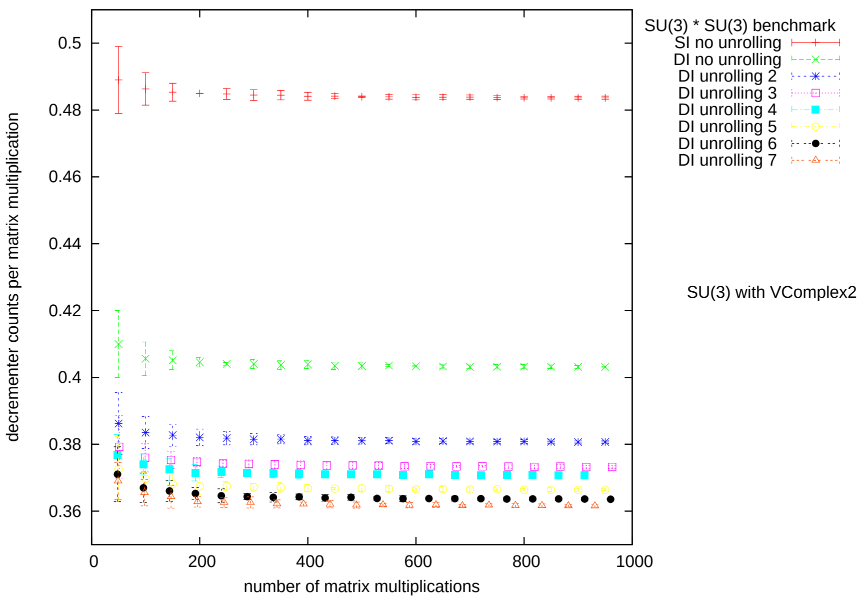
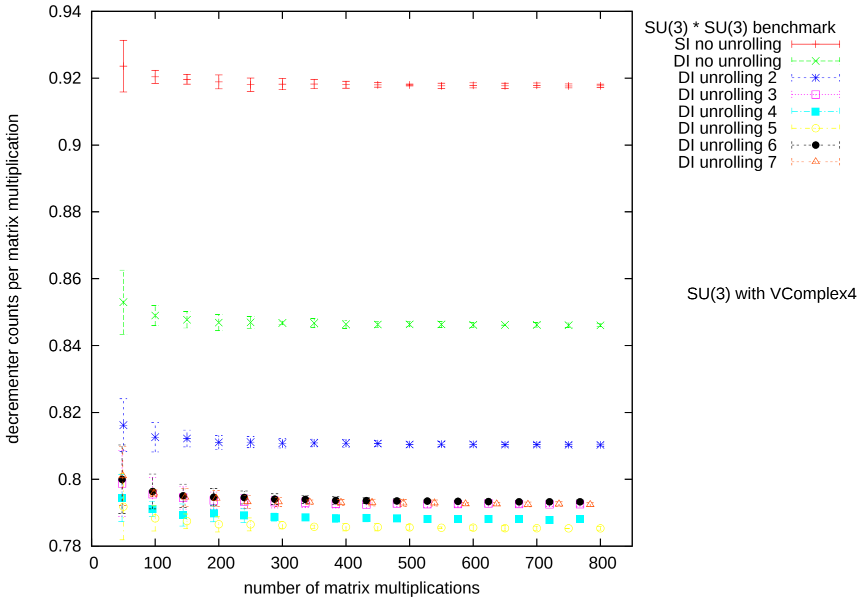


Figure 3: Loop benchmark test with NOPs



(a) VComplex2 implementation



(b) VComplex4 implementation

Figure 4: SU(3)·SU(3) benchmark test with the VComplex2 and VComplex4 implementation