28-03-2022

D3.2 – Initial public release of high-performance software components

Version 1.0

GA no 952165

Dissemination Level

☒ PU: Public
☐ PP: Restricted to other programme participants (including the Commission)
☐ RE: Restricted to a group specified by the consortium (including the Commission)
☐ CO: Confidential, only for members of the consortium (including the Commission)

Document Information

Project Title	Targeting Real chemical accuracy at the EXascale
Project Acronym	TREX
Grant Agreement No	952165
Instrument	Call: H2020-INFRAEDI-2019-1
Topic	INFRAEDI-05-2020 Centres of Excellence in EXascale computing
Start Date of Project	01-10-2020
Duration of Project	36 Months
Project Website	https://trex-coe.eu/
Deliverable Number	D3.2
Deliverable title	D3.2 – Initial public release of high-performance software components, as seen in GA
Due Date	M06 – 31-03-2021 (from GA)(from GA)
Actual Submission Date	28-03-2022
Work Package	WP3 – Code modularization and interfacing
Lead Author (Org)	William Jalby (Université de Versailles Saint-Quentin-en-Yvelines (UVSQ))
Contributing Author(s) (Org)	Vijay Gopal Chilkuri (Centre National de la Recherche Scientifique (CNRS)), Anthony Scemama (CNRS)
Reviewers (Org)	Fabio Affinito (CINECA), Matthias Rupp (UKON)
Version	1.0
Dissemination level	PU
Nature	Report
Draft / final	Final
No. of pages including cover	vii (front matter), 20 (main text), VIII (back matter)

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Disclaimer

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Versioning

Version	Date	Authors	Notes
1.0	28-03-2022	William Jalby (UVSQ), Anthony Scemama (LCPQ), Vijay Gopal Chilkuri (LCPQ)	First Official Release

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Abbreviations

ASM	inline assembly
AVX	advanced vector extensions
BLAS	basic linear algebra subprograms
BLIS	BLAS-like library instantiation software framework
CNRS	Centre National de la Recherche Scientifique
FMA	fused multiply-add
HPC	high performance computing
IR	intermediate representation
ISA	instruction set architecture
LLVM	Low Level Virtual Machine
MAQAO	Modular Assembly Quality Analyzer and Optimizer
MIPP	MyInstrinsics++
MKL	math kernel library
QMC	quantum Monte Carlo
SSE	streaming single instruction multiple data (SIMD) extensions
SIMD	single instruction multiple data
SVE	scalable vector extensions
UVSQ	Université de Versailles Saint-Quentin-en-Yvelines

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Table of Contents

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

List of Figures

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

1 Introduction

Besides development of the TREX falgship codes, Quantum Monte Carlo kernel library (QMCkl) development is a major effort in the TREX project, spanning the whole project duration. Four deliverables are devoted to cover this development. First delivered at Month.6,, D1.1 "Software Design and implementation of QMCkl" described the definition of the API and the "pedagogical" presentation of the algorithm heavily based on Literate Programming. Then at Month 9, D1.2 "Report on pre release of open source readable implementation of QMCkl" presented some very general guidelines for QMCkl, essentially focusing on the library usage. Also in D1.2, the major kernels were presented from an algorithmic point of view and how such kernels could be used in TREX applications. Finally, D1.2 introduced the idea of Control of Numerical Accuracy and the corresponding implementation Verificarlo CI. Then D3.2 (delivered at Month 18) and D3.4 (to be delivered at Month 36) are focused on the HPC side of the library. This current deliverable D3.2 will present the main rationale behind the development of the HPC components as well as some first implementation results while D3.4 will detail the final status of the library together with an extensive performance test.

In this first document, after a brief Introduction (Section 1), we are describing the rationale for QMCkl design guidelines (Section 2). In particular, objectives and general methodology will be highlighted. Then, more precise details on the methodology will be given by walking through a detailed DGEMM example (Section 3). Finally, some preliminary performance results on a key QMCkl routine (JASTROW computation will be given).

This first document essentially covers work in progress. As such, first, many implementation choices presented in this first document are still preliminary and might evolve. Second, we first focused on multicore CPU: although we kept in mind GPU constraints and issues, most of the implementation efforts targeted multicore CPU (X86 and ARM) and the first performance results have been obtained on X86. More performance tests on X86 are under way, tests on ARM will be carried out in the coming months as well as fitst QMCkl extensions to cover GPU. All of these topics will be covered in detail in the second deliverable.

QMCKL is publicly available (source code and documentation): the source code is available on the Github repository at `https://github.com/TREX-CoE/qmckl` and the documentation can be found at `https://trex-coe.github.io/qmckl`.

2 QMCkl rationale

2.1 The stage/context

The hardware and software for high performance computing (HPC) systems is still evolving but some main trends are emerging as stable for the next ten years: increased importance of parallelism (intra-node, inter-node, accelerator) and data access (complex memory hierarchies, communication between nodes/accelerators). For materials science simulations quantum Monte Carlo (QMC) methods offer very high level accuracy at the expense of very high computational costs. Fortunately,

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

QMC methods are embarrassingly parallel (due to their Monte Carlo nature) with limited memory requirements making such methods very attractive and competitive on modern high performance systems.

The primary goal of QMCkl library is to provide QMC code developers with a set of routines fulfilling a "triple objective":

1. Portability: the library must be available across different hardware and software environments
2. Performance: the library must provide a very good (not necessary optimal) level of performance on different hardware/software
3. Productivity: the library must be easy to use and to integrate for QMC developers (i.e. physicists/chemists with a limited background in Computer Science)

This triple "objective" is very challenging but classical and very often claimed. It has been set forward in much a larger setting for whole applications. Supercomputing Conference has started a series of workshops on this broad topic: Performance, Portability and Productivity in HPC.[1] Beyond this triple objective, we will try to address important issues such as energy consumption and numerical accuracy. In our case (library design), we advocate that such an ambitious goal can be reached by an appropriate methodology design and a goal reformulation. Additionally, using QMCkl should allow the scientific development to focus on the scientific part of the code, the HPC being taken care of by QMCkl.

First, we decouple portability from performance: for QMCkl the first goal is to provide correct results and therefore QMCkl will be provided in an easy to port/integrate version which will allow the code developer to test QMCkl use. Along this line, QMCkl will provide tools for checking numerical stability (Verificarlo).

Second for performance, we will not go after a universal goal of high performance on any arbitrary system. We will structure different hardware targets in a hierarchy privileging clearly a few systems (called reference architectures). On such architectures, performance optimization efforts will be carried out resulting in a specific code for each architecture and correspondingly a specific set of tools for fine tuning. Additionally, it should be noted that not only hardware differences have to be dealt with but also different software environments (in particular compilers) have to be supported.

Finally, for productivity our primary objective will be to ensure that for our reference TREX codes, at least 50% of execution time is spent in QMCkl. This will require on one hand that the library provides enough functionalities to cover a large part of application needs and on the other hand that the code developer makes some effort to integrate QMCkl routines. It should be noted that our simple metrics (50 % of execution time) can be easily perverted or biased but in our case since we are controlling TREX code, we can make sure that such a metric is sill meaningful. This simple metric is also meant as an internal guide only. There are many more aspects to productivity, most of

[1] https://p3hpc.org/

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

them difficult to quantify. It should be noted that the development of each routine in two flavors: a pedagogical one and HPC variants is directly intended to favor productivity in making QMCkl easy to use.

2.2 Key guidelines for QMCkl design

In this section we will detail the major guidelines which are driving QMCkl development.

2.2.1 Focus and specialization

The primary goal of QMCkl is to support QMC applications and therefore we will use key characteristics of these applications. For example, the fact that most QMC applications contain a high degree of parallelism (based on Monte Carlo computation principles) will drive most of the QMCkl effort towards developing an efficient single node version, the multinode exploitation being taken care of by running multiple copies of QMCkl.
Similarly, we will restrict in a first phase our optimization effort towards "reference CPUs": Intel Xeon line, AMD Zen line and ARM Neoverse (N and V lines). This choice is driven by the fact that these CPUs represent the largest share in HPC market systems. It should be noted that we are not selecting specific processors but rather processor lines with multiple implementations in the next ten years. Our objective will be to make sure that we are able to follow CPU evolutions during the upcoming decade.
As mentioned in the introduction, our major effort on GPU will come in a second phase. Having a bit delayed our major effort on GPU will provide us two interesting side benefits. First although the NVIDIA line dominates the market, upcoming GPU from Intel and AMD are interesting to consider and we will wait until 3 types of GPUs are available. Second, the current GPU generation induces numerous and costly memory transfers (offloading) between CPU memory and GPU memory. For the moment, the cost of these exchanges strongly limits the GPU applicability scope.
Also we will focus on the specific needs of our target applications: for example, QMC codes heavily use matrix multiplications; however, by analyzing further, it appears that most of the matrix multiplications are rank K updates: multiplying a tall matrix $(N \times K)$ by a flat matrix $(K \times N)$ with K typically less than 32 and much smaller than N values (from 100 to a few thousands). Therefore, instead of optimizing a general DGEMM routine, we will develop a rank K update specially tuned for taking advantage of the peculiar matrix sizes and shapes.

2.2.2 Ease of interaction with software and hardware environment

Analyzing the current HPC implementation of basic linear algebra subprograms (BLAS) library such as Intel's math kernel library (MKL), we can clearly see that the binary MKL structure prevents optimizations between a library subroutine and its environment: for example, if there are consecutive calls to DAXPY routines sharing arrays, MKL and the compiler will miss that optimization opportunity. The jitted version of DGEMM MKL for small matrices (see https://github.com/libxsmm/libxsmm) allows some (limited) adaptation to the calling parameters. Essentially the jitting amounts to generate on-the-fly specific library versions for small matrix sizes: however, there is a non-negligible cost for generating the first version, this cost has to be amortized across several calls with exactly

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

the same parameters. Therefore, there is a clear lack of flexibility. In QMCkl, flexibility will be achieved by providing a source-level optimized version of the library, allowing the compiler to perform optimizations across library boundaries. This feature will be essential for routines which are using $\mathcal{O}(N)$ data and performing $\mathcal{O}(N)$ operations. This extra flexibility has to be exploited with some care to avoid downsides such as code bloat. For controlling that aspect, we will use specific tools such as CQA (Code Quality Analyzer) to monitor code quality

In developing optimized versions, specialized versions with respect to the different reference architectures will be developed. Furthermore, specific auto-tuning tools will be used to get the "last mile" optimization providing typically 5 to 10% performance gain. For example, such optimizations will be particularly useful to exploit different number of registers availalble.

One of the most important hardware parameters to be exploited for efficient optimization will be memory hierarchy parameters (size, bandwidth and latency of each level). In order to address this specific issue, QMCkl will restructure 2D (and 3D) arrays in tiled versions: this point is presented in more details in Section 3. The address computations with such tiled arrays wil be more complex (accessing 4/5D arrays) but by adjusting tile size, use of memory hierarchy can be finely tuned. Finally, QMCkl will investigate further optimization goals than the classical "execution time". Since QMC kernels are extremely compute intensive, another optimization criterion will be energy consumption and with respect to that goal, specific code versions will be developed. Last but not least, all of the code will be Open Source allowing QMCkl users to customize routines to their precise needs.

2.2.3 Use of advanced tuning tools

Many modern libraries have directly embedded software tools (auto tuners) together with standard library routines. In some cases, such as ATLAS [1] or FFTW [2], software tools are mainly used for parameter tuning such as finding appropriate block size. In other cases, such as SPIRAL [3], software tools are used to generate automatically new kernel variants to be used as building blocks later.

In QMCkl, we will use software tools with similar goals such as generating new kernel variants and also fine tuning but we will also use tools to analyze library behavior and customize a few routines for specific needs.

First, we will use advanced performance analysis tools such as Modular Assembly Quality Analyzer and Optimizer (MAQAO)[2] [4] to analyze performance bottlenecks and explicit relations between algorithms and hardware.

Second, the importance of numerical accuracy should not be overlooked: for example, use of Single Precision instead of Double Precision can provide substantial performance gain. However, such gains should not be obtained at the cost of substantial accuracy loss. In fact, numerical accuracy needs to be monitored in order to keep it within acceptable limits. For that task, we will

[2] https://www.maqao.org

use Verificarlo[3] [5] which will allow us to detect configurations where numerical accuracy can accomodate mixed precision. More details on the use of Verificarlo in QMCkl are given in deliverable D1.2.

Third we will use systematically auto tuning tools for generating the most optimized version of a library routine for a given architecture. We will not use metaprogramming tools such as BOAST[6] because ultimately they rely on compilers which are difficult to control. Instead we will use assembly code templates with specific rules for unrolling and blocking and the auto tuner will modify directly these templates. Using directly Assembly Code will allow to bypass compilers and to achieve some insentivity to compiler quality.

2.3 Current hardware/software offering

There is a recent (December 2021) comparison of the latest offerings from Intel, AMD and ARM in terms of HPC processors.[4]

The latest cores from these three manufacturers share a strong set of characteristics: out-of-order execution, large number of functional units, vector units and multilevel memory hierarchies. Clearly, there are differences in implementation and in some key characteristics (size) of the various out-of-order buffers. Interestingly, most of these differences are well beyond current optimization capabilities even for state-of-the-art compilers: for example, the compiler has a hard time to generate a code which will be able to fully exploit the full size of the reservation stations and therefore offer a substantial performance gain.

Therefore, such differences will be a priori not exploited in the first optimization step. Instead we will focus on "key" differences in instruction set architectures (ISAs). To simplify the analysis, we will restrict our effort to instructions which are essential to QMCkl library (Vector and Scalar load/store, Vector and Scalar FP operations). Below are listed some of the major differences that will need specific optimization in QMCkl:

- Number of Scalar and Vector registers
- Width of Vector instructions
- Specific combined operations: x86 ISA allows to combine in a single instruction a Load and an Integer/FP operation. All in all this allows to save the use of one register.
- In x86 ISA, Vector fused multiply-add (FMA) operates only between vector registers while ARM scalable vector extensions (SVE) allows to use operand from a scalar register.
- In AVX512 there is no reduction instruction on a vector register

[3]https://github.com/verificarlo/verificarlo

[4]https://fuse.wikichip.org/news/4795/arm-launches-new-neoverse-n2-and-v1-server-cpus-1-4x-1-5x-ipc-sve-and-armv9/

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

All of these differences can be handled by specific tools which allow to generate an ARM version from an x86 version. More details on our views on a generic ASM able to encompass both ARM and X86 will be given in the subsequent sections.

2.4 Overall organization of QMCkl libraries

Each QMCkl routine will first consist of an easy to understand/manipulate code version (called pedagogical/reference code) and then several HPC code variants (typically one per target architecture). Such a strategy of having one HPC code variant per architecture is usable for libraries (while it is much more difficult for full applications). However, the generation of multiple variants should be as much as possible automated to make sure that library maintenance costs remain reasonable.

All of these HPC code variants will remain in a high-level language format with a few inner kernels using ASM (`asm volatile` mechanisms in C). In Section 3, detailed examples of such code will be given. Using such "low level" constructs will still allow the compiler to inline QMCkl calls and perform global optimization (without being stopped by library "boundaries"). Using ASM will be extremely useful to overcome compiler limitations and also to provide uniform code (and therefore uniform performance) across several compilers.

2.4.1 Pedagogical/reference version

This version should be kept as simple and concise as possible: on one hand, no loop unrolling or explicit transformations such as loop blocking should be present in this code version.

On the other hand, attention should be paid to favor vectorization at the innermost loop level. In particular at this innermost level:

- As much as possible, no branches or subroutine calls (the latter ones should be removed by using inlining)
- Loop nests should be restructured in such a way that the innermost loop has the largest number of iterations
- Arrays should be restructured in such a way that innermost generate stride 0 or stride 1 access

If the loop is supposed to be vectorizable, OpenMP pragmas for vectorization should be inserted (see https://www.openmp.org/spec-html/5.0/openmpsu42.html). Attention should be paid to provide the compiler with additional information through the clause added to the directive.

2.4.2 HPC Variants

A typical QMCkl routine will conist between ten and a few hundreds of lines of source code. The HPC optimizations (and the HPC variants) will focus on the most time consuming loop nests. The HPC variants will be essentially generated at the loop nest level. It is very important to analyze globally loop nests in order to exploit at best all optimization opportunities. For the HPC variants

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

we will have several schemes with increased complexity depending upon the importance and the nature of target loop nest.

Here, we will describe the most elaborate and complex one, which is structured in three levels. Some simpler schemes can by obtained by reducing the number of levels. Moving from one level to the next should be carried out automatically or at least semi-automatically:

- HIGH LEVEL (common to all target architectures). At the highest level, the algorithm should be written in standard C with performance critical parts written in a specific language supporting vector constructs. In our case, we are using an abstract ASM language encompassing both ARM and x86 ISA but other choices could be made such as MyInstrinsics++ (MIPP)[5] or Low Level Virtual Machine (LLVM) intermediate representation (IR)[6]. At this level, optimization beyond vectorization should remain minimal. Number of vector registers and register width should remain generic. Similarly, the vectorized form should not contain any explicit unrolling but provisions for performing unrolling such as unrolling templates should be provided.
- MID LEVEL (specific versions are generated for different architectures). There, different sets of building blocks are generated, one set per architecture. Typically, a set will contain the versions corresponding to different unrolling levels.
- LOW LEVEL: A final version combining the various building blocks will be generated through auto tuning.

2.4.3 A few characteristics of this generic ASM/assembly language

Here we list the main characteristics of the instructions constituting this generic ASM. Here the term generic means that we want to be able to generate from this ASM either x86 instructions or ARM instructions. Next section will give detailed example of use.

- Vector instructions will be parametrized by their length VL (expressed in elements) so the code should be generic and cover all possible Vector Lengths
- All instructions will be not differentiated with respect to Double Precision (DP) versus Single Precision (SP). It will be the optimizer which will take care of that specialization.
- All arithmetic instructions will operate between registers (ARM-like): no operand coming from memory can be used in an arithmetic instruction.
- All arithmetic instructions will use only Vector Registers as main operands no scalar registers can be used. This will require the use of Broadcast instructions to promote a scalar to vector.
- We will use a Vector Reduction (denoted VRED) instruction capable of summing all of the components of a vector register. Such an instruction is available in ARM ISA but currently it requires a chain of between 4 and 6 instructions on x86

[5]`https://github.com/aff3ct/MIPP`
[6]`https://llvm.org/docs/LangRef.html`

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

- The code to be duplicated will be enclosed between curly brackets {}. Right at the beginning, the order of duplication is specified: D1 means duplicate first then proceed with D2 and so on. The variable $ID1 (resp $ID2, etc...) will refer to the duplication index and will take consecutive integer values from 1 up to Duplication degree.

- Registers should be explicitly named to avoid any issue/limitations by compiler

COMMENT 1: This ASM could be based on an extension of Vector Coding tool such as MIPP. MIPP is a portable wrapper for SIMD instructions written in C++11. It supports Neon, streaming SIMD extensions (SSE), advanced vector extensions (AVX) and AVX-512.

COMMENT 2: For the moment, we are using a somewhat simpler ASM without the template for unrolling. This will be added in future releases.

2.4.4 Routine categories

We will use two important classifiers for QMCkl routines:

1. the array structures used: 1D, 2D, 3D. This first criteria will associate with each routine the various array structures used by the routine.

2. the amount of FP operation per data: we will be mostly interested in distinguishing the streaming kernels, for which less than a single FP operation is performed per data accessed, from kernels performing multiple operations per data, the latter ones being good candidates for locality optimization

3 Demonstrative application

3.1 Hierarchical Data layout

The layout of the input data is organized into Blocks and Tiles as described in detail in the BLAS-like library instantiation software framework (BLIS) papers [7, 8]. The blocking scheme takes care of the cache hierarchy whereas the tiling scheme is adopted for the particular BLAS micro-kernel which actually performs the algebraic manipulations. A matrix in our representation therefore becomes a 6-index tensor:

$$A(i,j) = A'(o,p,m_{\text{tile}},n_{\text{tile}},l_{\text{block}},k_{\text{block}}) \tag{1}$$

where the notation is in Fortran format, i.e., one-based column major notation with the largest dimension indices at the end. Here, the blocks are denoted by the dimensions $l_{\text{block}}, k_{\text{block}}$ and are represented by the `(MC,KC)` blocks in Figure 1. The tiled dimensions are represented by the indices $m_{\text{tile}}, n_{\text{tile}}$ and correspond to the `(MR,NR)` tiles represented by the red/violet/green thick lines in Figure 1. Linear algebra manipulations are done in the micro-kernels and only require data in the tiles therefore ensuring perfect data locality for near zero cache misses. All micro-kernels therefore are based on manipulation of data in local tiles only.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

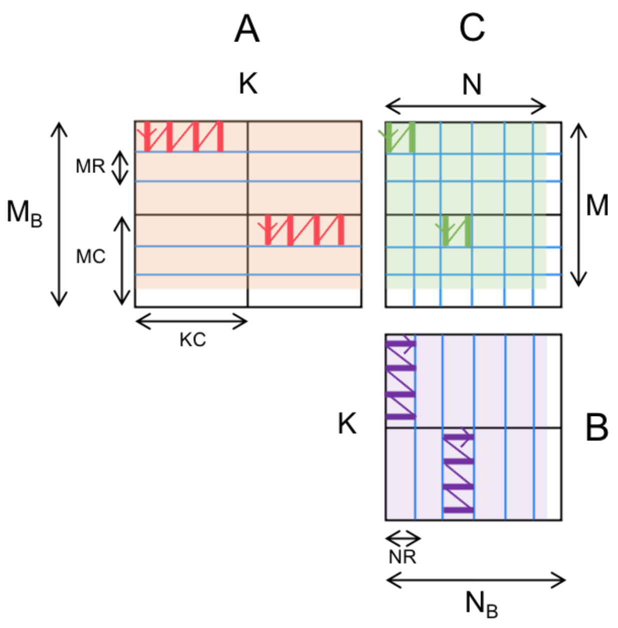


Figure 1: The internal representation of matrix data. The matrices are stored as blocks of dimension `(MC,KC)`. The blocks are further split into tiles of dimension `(MR,NR)` along only one dimension.

Note that the dimensions of the matrices represented internally are always multiples of `MR` and `NR` the choice of which will be explained below. Therefore, for input matrices whose dimensions are not multiples of `MR` and `NR`, the internal buffers are initialized by dimensions M_B and N_B which are the corresponding smallest multiples of `MR` and `NR` larger than or equal to M and N respectively. In this manner, matrices of general sizes can be represented using the aforementioned hierarchical data structure.

3.1.1 Optimized Cache Access

The data is accessed in a sequential and hierarchical manner as shown in Figure 2 so as to minimize the latency. The dimensions of the blocks `(MC,KC)` are chosen according to the L2 and L1 memory layout on the hardware. The L1 cache contains one tile of A, one tile of B, and one tile of C, whereas the L2 cache contains two blocks corresponding to A and B and one tile of C.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

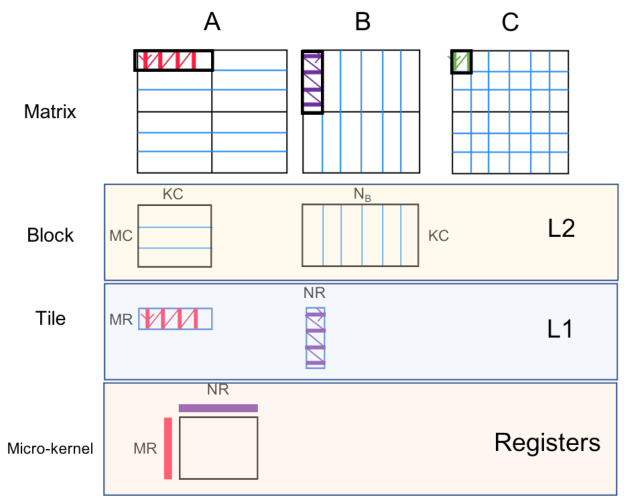


Figure 2: The hierarchical data layout adapted for the representation of matrices. The dimensions of the blocks $\mathtt{MC}, \mathtt{KC}, \mathtt{NC} = N$ and those of the tiles (`MR`,`NR`) are chosen such that they are adapted for the memory layout of the hardware.

3.2 Micro-kernel Design

3.2.1 Definitions

The kernel is defined in terms of fundamental operations involving vector registers (denoted as `VR`) which are of length `VL` and/or scalar registers (denoted as `SR`). Such an abstraction allows for an architecture independent description of the kernel and the key algorithms. An example of the `VR` for the AVX2 architecture is `YMM` with a vector length `VL`= 256 bits/4 DP elements, i.e. four double precision floating point values. The fundamental operations using these `VR`s are of two types:

- Load Store operations - These operations involve memory access and loading/storing of the data to/from `VR`s. The data can be loaded from/to a `VR` or a `SR`.
- Arithmetic operations - Operations such as `Op` = `ADD`/`MUL` involving `VR`s or `SR`s. Note that in some cases, such operations can also involve a memory address denoted by `MEM`.

Memory Operations

Most memory operations are of three types:

1. Loading data to a register - Denoted as `LOAD(VR, MEM)` or `LOAD(SR, MEM)`
2. Storing data to a memory address - Denoted as `STORE(MEM, VR)` or `STORE(MEM, SR)`
3. Broadcasting a scalar to a register - Denoted as `BROADCAST(VR, MEM)`
4. Prefetching instruction - Denoted as `PREFETCH(MEM)`

Here the operands follow Intel-based ordering, i.e. the result is always substituted to the left-most variable.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Arithmetic Operations

There are many arithmetic operations in the combined x86 and ARM instruction set, but here we shall focus on a limited subset which can be separated into four types:

1. Vector operations involving two operands - $\texttt{Op}(\texttt{MEM}, \texttt{VR}/\texttt{SR})$
2. Vector operations involving three/four operands - $\texttt{Op}(\texttt{VR1}, \texttt{VR2}, \texttt{VR3})$
3. Operations involving mixed operands - $\texttt{Op}(\texttt{VR1}, \texttt{VR2}, \texttt{SR3})$
4. Operations involving a memory operand - $\texttt{Op}(\texttt{VR1}, \texttt{VR2}, \texttt{MEM})$

Here, we describe the four operations in details. The first type of operation $\texttt{Op}(\texttt{SR1}, \texttt{VR1})$ (such as vector reduction $\texttt{ADDV}(\texttt{SR1}, \texttt{VR1}))$ applies an operator (addition) to all the elements of VR1 and stores the result in a scalar register SR1. An example of this type of operator is the horizontal Vector Reduce operation available in the RISC-based ARM instruction set. The second type of operation is purely vectorial, i.e. performs the operation on two VRs and stores the result in VR1. An example of this type of operation is the standard FMA $\texttt{VFMADD231PD}(\texttt{VR1}, \texttt{VR2}, \texttt{VR3})$ which multiplies the values in VR2 and VR3 and adds the result to VR1. The third type of operation is also based on ARM architecture and involves a mixed operation with a scalar value extracted from VR3 (determined by the lane value l) and the vector register VR2 and storing the result in a vector register VR1. Such an instruction can perform scaling of the elements of VR2. An example of such an operation is the $\texttt{FMLA}(\texttt{VR1}, \texttt{VR2}, \texttt{VR3}(\texttt{l}))$ which multiplies the vector elements of VR2 by the specified element from $\texttt{VR3}(l)$ and adds the result to VR1. Finally, the last operation involves a combined LOAD and artithmetic operation. VL double precision values are loaded directly from memory to perform the vector operation with a VR2 and store the result in a VR1.

3.3 Example kernels

3.3.1 DGEMM

The basic DGEMM expression shown in Eq 2 is modified in our block-tiled representation. This new algorithm is based on the Eq 2, which in our block-tiled format reads Eq 3.

$$C(i,j) = \sum_k A(i,k) \times B(k,j) \tag{2}$$

$$\begin{aligned} C(i,j) &= C'(o_c, p_c, l_c, k_c, m_c, n_c) \\ A(i,k) &= A'(o_c, p_c, l_c, k, m_c, n_c) \\ B(k,j) &= B'(o_c, p_c, k, k_c, n_c, n_c) \\ C'(o_c, p_c, l_c, k_c, m_c, n_c) &= \sum_k A'(o_c, p_c, l_c, k, m_c, n_c) * B'(o_c, p_c, k, k_c, m_c, n_c) \end{aligned} \tag{3}$$

The core of the computation show in Eq 3 involves the calculation of the outer product as shown in Figure 2. This outer product is carried out via a custom kernel which is written with inline assembly

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Algorithm 1 Micro-kernel DGEMM algorithm

```
Require: KC ≠ 0
  k ← 1
  for k ← 1 to KC do
      VR1  ← VLOAD(A(0, k))
      VR2  ← VLOAD(A(0+VL, k))
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                    ▷ FMA on first pair of Bs
      VR5  ← VFMA(VR5, VR1, VR3)
      VR6  ← VFMA(VR6, VR2, VR3)
      VR7  ← VFMA(VR7, VR1, VR4)
      VR8  ← VFMA(VR8, VR2, VR4)
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                    ▷ FMA on second pair of Bs
      VR9  ← VFMA(VR9 , VR1, VR3)
      VR10 ← VFMA(VR10, VR2, VR3)
      VR11 ← VFMA(VR11, VR1, VR4)
      VR12 ← VFMA(VR12, VR2, VR4)
      VR3  ← VBROADCAST(B(1, k))
      VR4  ← VBROADCAST(B(2, k))                    ▷ FMA on last pair of Bs
      VR13 ← VFMA(VR13, VR1, VR3)
      VR14 ← VFMA(VR14, VR2, VR3)
      VR15 ← VFMA(VR15, VR1, VR4)
      VR16 ← VFMA(VR16, VR2, VR4)
      k ← k+1
  end for
```

statements (`asm volatile`) in order to achieve the best possible execution. Pseudo-code of the micro-kernel is shown in Algorithm 1.

This kernel for DGEMM uses all 16 registers available on the AVX2 hardware. The `VR` represents a vector register and `VL` the vector length.

The data required for the sequence of operations in Algorithm 1 is streamed through the custom data structures A', B', and C' and ensures that subsequent tiles required for each step of the loop in Algorithm 1 are accessed in stride 1, i.e. via automatic hardware prefetch. The above kernel is called for all the tiles and blocks of the matrix C' and if required the subsequent result is unpacked in LAPACK format.

Preliminary results

The above kernel has been implemented for the `AVX2` and `AVX512` architectures. Comparison of the kernel for the `AVX2` and `AVX512` architecture with the corresponding optimal subroutines availble in the Intel Math Kernel Library (MKL) has been carried out. Preliminary results are shown in Figure 3 and Figure 4. A clear improvement is seen for all cases, especially for small matrices ($M = N < 1000$) on

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

the AVX512 machines. Further improvement involves preparing easy installation scripts and specialized subroutines for easy use in all QMCkl applications.

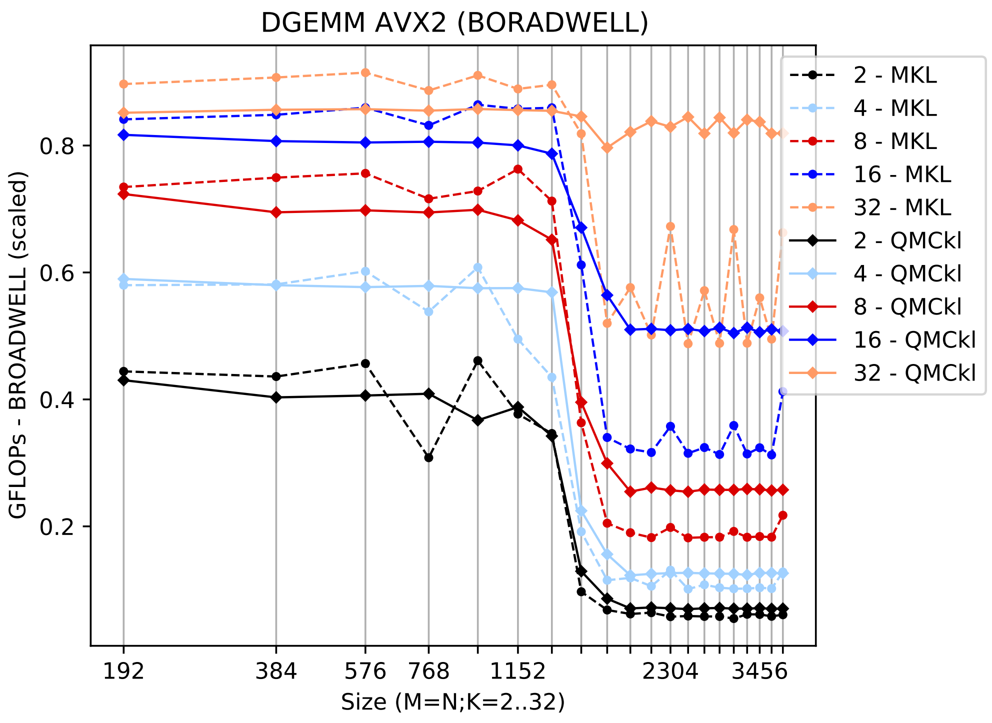


Figure 3: Comparison of the DGEMM performance for Rank-K updates vs Intel MKL on AVX2 (Intel BROADWELL) architecture.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

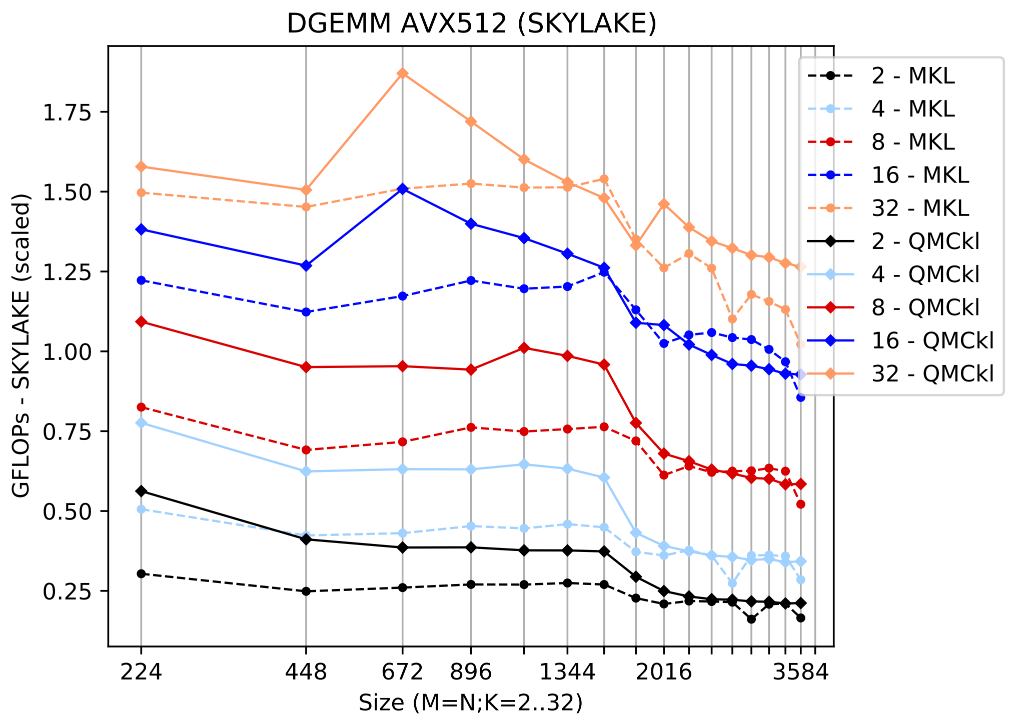


Figure 4: Comparison of the DGEMM performance for Rank-K updates vs Intel MKL on AVX512 (Intel SKYLAKE) architecture.

3.3.2 Vector Reduction

The Vector reduction instruction is sometimes needed for special kernels. Native instructions are available for some architectures such as ARM (RISC-V). The Intel x86 instruction set does not have a vector reduce (`ADDV`) instruction for a `VR`. In such a case, the `ADDV` can still be performed albeit at a higher cost with the following 5-set of instructions.

Algorithm 2 Micro-kernel `VRED`

```
XMM0 ← VCASTPD128(VR1)
XMM1 ← VEXTRACTF128PD(XMM0, 0x1)
XMM2 ← VADDPD(XMM0, XMM1)
XMM3 ← VUNPACKPD(XMM2, XMM2)
XMM0 ← VADDSD(XMM3, XMM2)
C(0) ← VSTORESD(XMM0)
```

Second type using only `YMM` registers
Where the meaning of each instruction is as follows:

- `VCASTPD128`: Take the low 128 bits of the `VR`

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

Algorithm 3 Micro-kernel VRED

```
VR2 ← VPERM2F128(VR1, VR1)
VR3 ← VADDPD(VR0, VR1)
VR4 ← VPERMILPD(VR0, 0x5)
VR5 ← VADDPD(VR3, VR4)
C(0) ← BROADCASTPD(VR5)
```

- VEXTRACTF128PD: Extract the high 128 bits of the VR
- VUNPACKPD: Extract the high 64 bits of the VR
- VPERM2F128: Permute the low 128 and high 128 bits of VR
- VPERMILPD: Permute the two 64 bit numbers within each 128 bit lane in VR

4 Preliminary performance analysis

In this section, we show performance of one of the most important QMCkl kernels: the computation of the Jastrow factor, for which the computation is fairly complex: over several hundreds of source code line. This kernel has already been presented in the reports of the Work Package (WP) in which the pedagogical version of the library is developed (WP1), but here we recall the equations and present its performance in the optimized version of QMCkl.

The initial equation implemented in CHAMP is:

$$J_{\text{een}}(\mathbf{r},\mathbf{R}) = \sum_{\alpha=1}^{N_{\text{nucl}}} \sum_{i=1}^{N_{\text{elec}}} \sum_{j=1}^{i-1} \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} c_{lkp\alpha} \left(r_{ij}\right)^k \left[\left(R_{i\alpha}\right)^l + \left(R_{j\alpha}\right)^l\right] \left(R_{i\alpha}\, R_{j\alpha}\right)^{(p-k-l)/2}$$

It was rewritten as

$$J_{\text{een}}(\mathbf{r},\mathbf{R}) = \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} \sum_{\alpha=1}^{N_{\text{nucl}}} c_{lkp\alpha} \sum_{i=1}^{N_{\text{elec}}} \bar{\mathtt{R}}_{i,\alpha,(p-k-l)/2}\ \bar{\mathtt{P}}_{i,\alpha,k,(p-k+l)/2}$$

with

$$\bar{\mathtt{P}}_{i,\alpha,k,l} = \sum_{j=1}^{N_{\text{elec}}} \bar{\mathtt{r}}_{i,j,k}\ \bar{\mathtt{R}}_{j,\alpha,l}.$$

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

The gradients and Laplacian of the Jastrow factor are also required:

$$\begin{aligned}\nabla_{im} J_{\text{een}}(\mathbf{r},\mathbf{R}) \;=\; & \sum_{p=2}^{N_{\text{nord}}} \sum_{k=0}^{p-1} \sum_{l=0}^{p-k-2\delta_{k,0}} \sum_{\alpha=1}^{N_{\text{nucl}}} c_{lkp\alpha} \sum_{i=1}^{N_{\text{elec}}} \bar{\mathtt{G}}_{i,m,\alpha,(p-k-l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k+l)/2} + \\ & \bar{\mathtt{G}}_{i,m,\alpha,(p-k+l)/2}\, \bar{\mathtt{P}}_{i,\alpha,k,(p-k-l)/2} + \bar{\mathtt{R}}_{i,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,m,\alpha,k,(p-k+l)/2} + \\ & \bar{\mathtt{R}}_{i,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,m,\alpha,k,(p-k-l)/2} + \delta_{m,4}\Big(\\ & \bar{\mathtt{G}}_{i,1,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,1,\alpha,k,(p-k-l)/2} + \bar{\mathtt{G}}_{i,2,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,2,\alpha,k,(p-k-l)/2} + \\ & \bar{\mathtt{G}}_{i,3,\alpha,(p-k+l)/2}\, \bar{\mathtt{Q}}_{i,3,\alpha,k,(p-k-l)/2} + \bar{\mathtt{G}}_{i,1,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,1,\alpha,k,(p-k+l)/2} + \\ & \bar{\mathtt{G}}_{i,2,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,2,\alpha,k,(p-k+l)/2} + \bar{\mathtt{G}}_{i,3,\alpha,(p-k-l)/2}\, \bar{\mathtt{Q}}_{i,3,\alpha,k,(p-k+l)/2}\Big)\end{aligned}$$

with

$$\bar{\mathtt{G}}_{i,m,\alpha,l} = \frac{\partial\,(R_{i\alpha})^l}{\partial r_i}, \qquad \bar{\mathtt{g}}_{i,m,j,k} = \frac{\partial\,(r_{ij})^k}{\partial r_i}, \qquad \text{and}\;\; \bar{\mathtt{Q}}_{i,m,\alpha,k,l} = \sum_{j=1}^{N_{\text{elec}}} \bar{\mathtt{g}}_{i,m,j,k}\, \bar{\mathtt{R}}_{j,\alpha,l}.$$

We can remark that reshaping the tensors into matrices by grouping indices allows us to rely on the DGEMM kernel for the most expensive pieces of the equations. Special attention was dedicated in the data structures to allow this reshaping with zero-copy, and to ensure a stride-one access to array in the computations that occur outside of the DGEMM kernels.

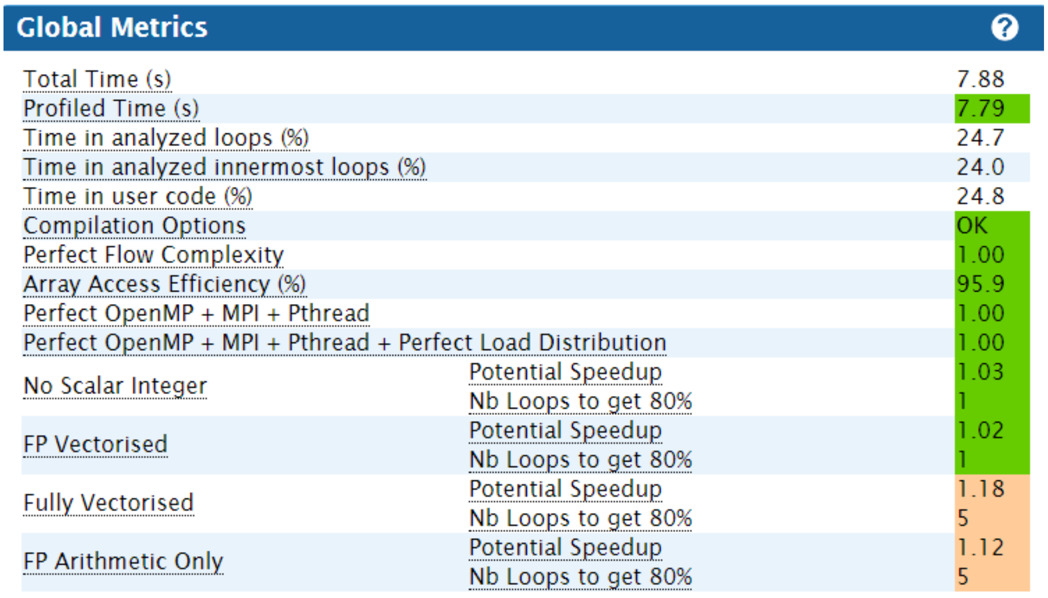
All of the measurements presented in this section have been carried out on a single core of an Intel Skylake Intel(R) Xeon(R) Platinum 8170 CPU @ 2.10GHz, running Linux 5.16.-arch1. QMCkl was compiled using the Intel Fortran compiler `ifort` 2021.5.0 with the `-O3` option. All performance analysis (measurements and data formatting) was done using MAQAO/ONEVIEW toolset.
Note that in the runs presented in this section, the kernels described in the previous section are not yet linked with the QMCkl library, and MKL is used for matrix multiplications. In the short term, we expect our kernels to improve performance.

Figure 6 shows that most (more precisely 76%) of the execution time is spent in math libraries. Figure 7 indicates that most of this library time is spent in DGEMM (Dense Matrix Multiplication) routines. For this initial run classic MKL BLAS was used but in subsequent runs, QMCkl DGEMM will be used.

Figure 5 shows that beyond libraries, 25% of the execution time is spent in loops and most of it in innermost loops. Still on Figure 5, the row "Perfect Flow Complexity" with a value of 1.00 demonstrates that all of these loops are branch free and call free. The row "Array Access Efficiency" with a value of 95.9 indicates that 95.9 % of data access in innermost loops are stride 1 access showing an excellent array organization and a very good use of spatial locality.

Finally, Figure 8 shows that many loops are very well vectorized: see green cells with 100 % in the column "Vectorization Ratio". However, the column "Vectorization Efficiency" with values under 50 clearly reveals that the compiler was not able to generate full 512 bits vector instructions. Most of the time, instructions using 256 bits vectors have been generated. This weakness deserves further investigation.

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.



Global Metrics

Total Time (s)		7.88
Profiled Time (s)		7.79
Time in analyzed loops (%)		24.7
Time in analyzed innermost loops (%)		24.0
Time in user code (%)		24.8
Compilation Options		OK
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		95.9
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	1.03
	Nb Loops to get 80%	1
FP Vectorised	Potential Speedup	1.02
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	1.18
	Nb Loops to get 80%	5
FP Arithmetic Only	Potential Speedup	1.12
	Nb Loops to get 80%	5

Figure 5: Global MAQAO metrics showing the overall performance of the Jastrow submodule of the QMCkl library.

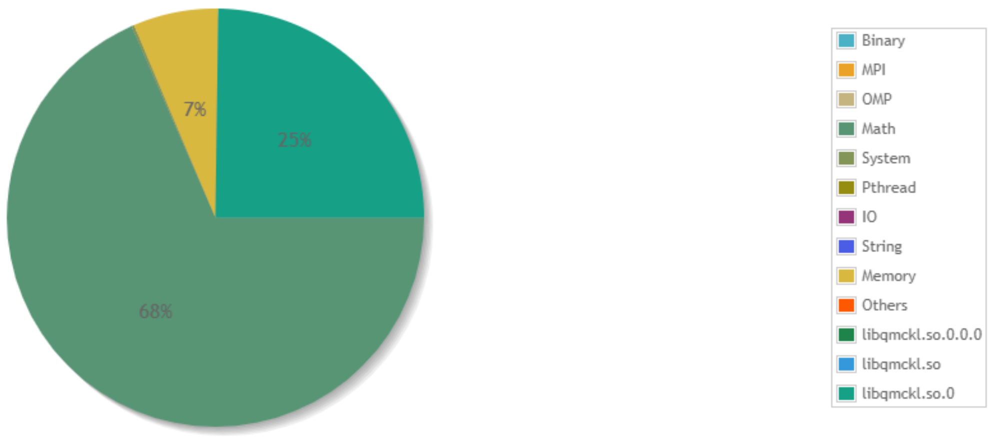
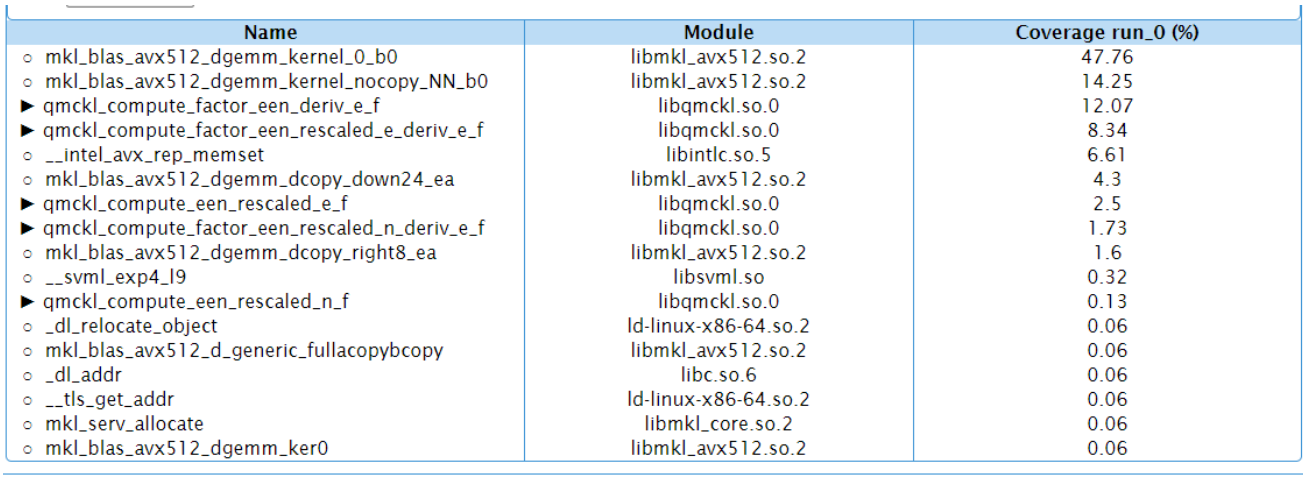
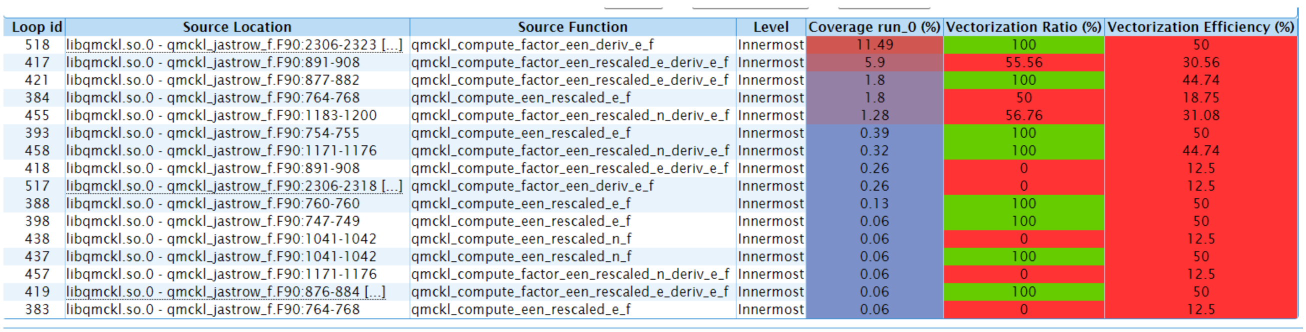


Figure 6: The percentage of time spent in the various types of operations. The majority of the time i.e. 68% is spent in the Math (i.e. MKL DGEMM) library.



Name	Module	Coverage run_0 (%)
○ mkl_blas_avx512_dgemm_kernel_0_b0	libmkl_avx512.so.2	47.76
○ mkl_blas_avx512_dgemm_kernel_nocopy_NN_b0	libmkl_avx512.so.2	14.25
▶ qmckl_compute_factor_een_deriv_e_f	libqmckl.so.0	12.07
▶ qmckl_compute_factor_een_rescaled_e_deriv_e_f	libqmckl.so.0	8.34
○ __intel_avx_rep_memset	libintlc.so.5	6.61
○ mkl_blas_avx512_dgemm_dcopy_down24_ea	libmkl_avx512.so.2	4.3
▶ qmckl_compute_een_rescaled_e_f	libqmckl.so.0	2.5
▶ qmckl_compute_factor_een_rescaled_n_deriv_e_f	libqmckl.so.0	1.73
○ mkl_blas_avx512_dgemm_dcopy_right8_ea	libmkl_avx512.so.2	1.6
○ __svml_exp4_l9	libsvml.so	0.32
▶ qmckl_compute_een_rescaled_n_f	libqmckl.so.0	0.13
○ _dl_relocate_object	ld-linux-x86-64.so.2	0.06
○ mkl_blas_avx512_d_generic_fullacopybcopy	libmkl_avx512.so.2	0.06
○ _dl_addr	libc.so.6	0.06
○ __tls_get_addr	ld-linux-x86-64.so.2	0.06
○ mkl_serv_allocate	libmkl_core.so.2	0.06
○ mkl_blas_avx512_dgemm_ker0	libmkl_avx512.so.2	0.06

Figure 7: A breakdown of the total time spent in the various function of the Jastrow submodule. As can be clearly seen, the majority of the time (>50%) is spent in the call to MKL DGEMM subroutine.



Loop id	Source Location	Source Function	Level	Coverage run_0 (%)	Vectorization Ratio (%)	Vectorization Efficiency (%)
518	libqmckl.so.0 - qmckl_jastrow_f.F90:2306-2323 [...]	qmckl_compute_factor_een_deriv_e_f	Innermost	11.49	100	50
417	libqmckl.so.0 - qmckl_jastrow_f.F90:891-908	qmckl_compute_factor_een_rescaled_e_deriv_e_f	Innermost	5.9	55.56	30.56
421	libqmckl.so.0 - qmckl_jastrow_f.F90:877-882	qmckl_compute_factor_een_rescaled_e_deriv_e_f	Innermost	1.8	100	44.74
384	libqmckl.so.0 - qmckl_jastrow_f.F90:764-768	qmckl_compute_een_rescaled_e_f	Innermost	1.8	50	18.75
455	libqmckl.so.0 - qmckl_jastrow_f.F90:1183-1200	qmckl_compute_factor_een_rescaled_n_deriv_e_f	Innermost	1.28	56.76	31.08
393	libqmckl.so.0 - qmckl_jastrow_f.F90:754-755	qmckl_compute_een_rescaled_e_f	Innermost	0.39	100	50
458	libqmckl.so.0 - qmckl_jastrow_f.F90:1171-1176	qmckl_compute_factor_een_rescaled_n_deriv_e_f	Innermost	0.32	100	44.74
418	libqmckl.so.0 - qmckl_jastrow_f.F90:891-908	qmckl_compute_factor_een_rescaled_e_deriv_e_f	Innermost	0.26	0	12.5
517	libqmckl.so.0 - qmckl_jastrow_f.F90:2306-2318 [...]	qmckl_compute_factor_een_deriv_e_f	Innermost	0.26	0	12.5
388	libqmckl.so.0 - qmckl_jastrow_f.F90:760-760	qmckl_compute_een_rescaled_e_f	Innermost	0.13	100	50
398	libqmckl.so.0 - qmckl_jastrow_f.F90:747-749	qmckl_compute_een_rescaled_e_f	Innermost	0.06	100	50
438	libqmckl.so.0 - qmckl_jastrow_f.F90:1041-1042	qmckl_compute_een_rescaled_n_f	Innermost	0.06	0	12.5
437	libqmckl.so.0 - qmckl_jastrow_f.F90:1041-1042	qmckl_compute_een_rescaled_n_f	Innermost	0.06	100	50
457	libqmckl.so.0 - qmckl_jastrow_f.F90:1171-1176	qmckl_compute_factor_een_rescaled_n_deriv_e_f	Innermost	0.06	0	12.5
419	libqmckl.so.0 - qmckl_jastrow_f.F90:876-884 [...]	qmckl_compute_factor_een_rescaled_e_deriv_e_f	Innermost	0.06	100	50
383	libqmckl.so.0 - qmckl_jastrow_f.F90:764-768	qmckl_compute_een_rescaled_e_f	Innermost	0.06	0	12.5

Figure 8: Analysis of the native functions of the library shows good vectorization ratio (close to 100%) for almost all functions.

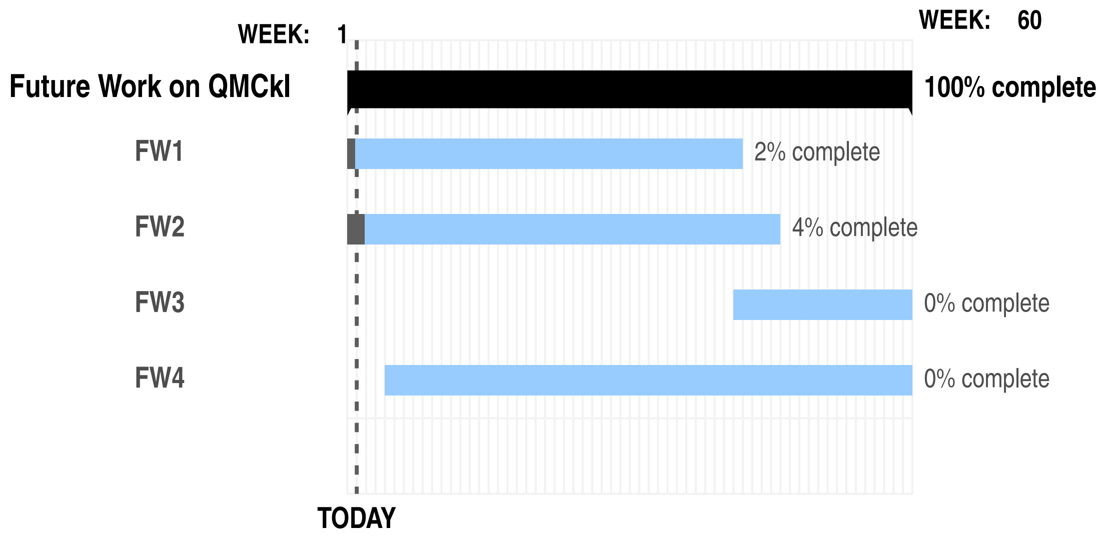
5 Conclusion and Future Work

In this document we presented the major guidelines and ingredients to generate the HPC versions of QMCKL. We demonstrated the use of these principles on generating a few kernels and testing one key QMCkl routine (Jastrow computation) on an x86 CPU.

We will pursue our work along 4 main directions:

- (**FW1**) Generate and test more QMCkl HPC routines
- (**FW2**) Integrate QMCkl_DGEMM into QMCkl main repository
- (**FW3**) Extend our work towards testing a larger number of compilers and hardware architectures (AMD, ARM and INTEL)
- (**FW4**) Integrate GPU support in QMCkl library

For sure, after going through all of the testing listed above, some of the concepts and implementations will have to be amended/corrected or perharps changed. As such, this work should be still considered as in progress.
Also in the project final year, once enough experience will have been accumulated with QMCkl, we will promote the library out side of TREX, potentially adding new kernels which could be essential to other QMC applications.



Acknowledgements

The authors would like to specially thank the two reviewers Fabio Affinito (CINECA) and Matthias Rupp (UKON). They carefully read the document making numerous comments which required some effort from our side but they contributed a lot to improve deliverable quality and readibility. Special thanks to Jan Beerens who kept us on track for deadlines.

References

[1] R. C. Whaley and A. Petitet, “Minimizing development and maintenance costs in supporting persistently optimized BLAS,” *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005, http://www.cs.utsa.edu/ whaley/papers/spercw04.ps. 2.2.3

[2] M. Frigo and S. G. Johnson, “The Design and Implementation of FFTW3,” *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Jan 2005. 2.2.3

[3] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “SPIRAL: Code Generation for DSP Transforms,” *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, Jun 2005. 2.2.3

[4] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby, “Maqao : Modular assembler quality analyzer and optimizer for itanium 2,” 2005. 2.2.3

[5] C. Denis, P. De Oliveira Castro, and E. Petit, “Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic,” in *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. IEEE, Jul 2016, pp. 55–62. 2.2.3

[6] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, “BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications,” *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 1, pp. 28–44, Aug 2017. 2.2.3

[7] K. Goto and R. A. v. d. Geijn, “Anatomy of high-performance matrix multiplication,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, pp. 1–25, 2008. 3.1

[8] F. G. Van Zee and R. A. Van De Geijn, “Blis: A framework for rapidly instantiating blas functionality,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, pp. 1–33, 2015. 3.1

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

A Appendix A

A.1 Unrolling and Register size - Kernel implementation

Algorithm 1 represents the high-level schematic of the kernel. The kernel itself is generated via a code generator written in Python which has the following features:

- Automatic loop unrolling - `asm volatile` code can be generated unrolled K times, taking advantage of the pipelining feature of the CPU.
- Kernel register type - The algorithm Algo 1 requires the knowledge of the number of physical registers and the register length (XMM, YMM, or ZMM). The number of registers and their type may vary between the different architectures, for example the INTEL (AVX2) instruction set has 16 physical YMM registers whereas the Intel (AVX512) instruction set has 32 physical ZMM type registers. The Python script is capable of automatically generating kernels for each such type of instruction sets.
- Kernel register usage - Moreover, the `asm volatile` kernel for the algorithm shown in Algo 1 can be written using a variable number of registers in order to optimize the data streaming on blocks of matrix A', B', or C'.

A.2 Code generator

The Python code given below is only 250 lines long. It shows the simplicity, versatility, and ease of maintainability of the script for future/new architectures.

```
beginning =r"""
  BEGIN_ASM()
"""

ending =r"""

    VZEROUPPER()

  END_ASM
  (
   : // output
   : // input
   [k] "m"(kl),
   [a] "m"(A),
   [b] "m"(B),
   [c] "m"(C),
   : // clobber
        "rax", "rbx", "rcx", "rdx", "rdi", "rsi", "r8", "r9", "r10", "r11", "r12",
          "r13", "r14", "r15", reglist "memory"
  )
"""

def initReg(numReg, sizeReg):
    regdict = dict()
    regdict[128] = 'XMM'
    regdict[256] = 'YMM'
    regdict[512] = 'ZMM'
    outcode =r""
    tmpcode =r""
    setregzero = "VXORPD(regname( idreg), regname( idreg), regname( idreg))"
    for i in range(numReg):
       tmpcode =           setregzero.replace("idreg",str(i))
       outcode = outcode + tmpcode.replace("regname",regdict[sizeReg]) + "\n"

    return(outcode)

def setupRegs():
    outcode=r""
    outcode = outcode + "MOV(RSI, VAR(k))" + "\n"
    outcode = outcode + "MOV(RAX, VAR(a))" + "\n"
    outcode = outcode + "MOV(RBX, VAR(b))" + "\n"
    outcode = outcode + "MOV(RCX, VAR(c))" + "\n"
    return(outcode)

beginning =r"""
  BEGIN_ASM()
"""

ending =r"""

    VZEROUPPER()

  END_ASM
  (
   : // output
   : // input
   [k] "m"(kl),
   [a] "m"(A),
   [b] "m"(B),
   [c] "m"(C),
   : // clobber
```

```
        "rax", "rbx", "rcx", "rdx", "rdi", "rsi", "r8", "r9", "r10", "r11", "r12",
          "r13", "r14", "r15", reglist "memory"
  )
"""


def initReg(numReg, sizeReg):
    regdict = dict()
    regdict[128] = 'XMM'
    regdict[256] = 'YMM'
    regdict[512] = 'ZMM'
    outcode =r""
    tmpcode =r""
    setregzero = "VXORPD(regname( idreg), regname( idreg), regname( idreg))"
    for i in range(numReg):
       tmpcode =           setregzero.replace("idreg",str(i))
       outcode = outcode + tmpcode.replace("regname",regdict[sizeReg]) + "\n"

    return(outcode)


def setupRegs():
    outcode=r""
    outcode = outcode + "MOV(RSI, VAR(k))" + "\n"
    outcode = outcode + "MOV(RAX, VAR(a))" + "\n"
    outcode = outcode + "MOV(RBX, VAR(b))" + "\n"
    outcode = outcode + "MOV(RCX, VAR(c))" + "\n"
    return(outcode)

def mainLoop(numReg, sizeReg, unrollFactor):
    regdict = dict()
    regdict[128] = "XMM"
    regdict[256] = "YMM"
    regdict[512] = "ZMM"
    regdictlc = dict()
    regdictlc[128] = "xmm"
    regdictlc[256] = "ymm"
    regdictlc[512] = "zmm"
    nelemInReg = sizeReg//64
    prefetch  = "PREFETCH(0, MEM(regname, idfac2*8))"
    loadmtor  = "VMOVUPD(REG( idreg1), MEM(regname, idfac2*8))"
    loadrtom  = "VMOVUPD(MEM(regname, idfac2*8), REG( idreg1))"
    add       = "VADDPD(REG( idreg1), REG( idreg2), MEM(regname, idfac2*8))"
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

```
broadcast = "VBROADCASTSD(REG( idreg1), MEM(regname, idfac2*8))"
fma       = "VFMADD231PD(REG( idreg1), REG( idreg2), REG( idreg3))"
lea       = "LEA(regname, MEM(regname, jmpfac*8))"
outcode =r""
finalcode =r""
for ik in range(unrollFactor):
    outcode =r""
    # Main loop
    NR = (numReg - 4)//2
    MR = 2 * (sizeReg//64)
    tmpload = loadmtor.replace("REG",regdict[sizeReg])
    tmpload = tmpload.replace("regname","RAX")
    tmpload = tmpload.replace("szereg",str(sizeReg))
    tmpload = tmpload.replace("idreg1",str(0))
    tmpload = tmpload.replace("idfac2",str(0))
    outcode = "\n" + "\t" + outcode + tmpload + "\n"
    tmpload = loadmtor.replace("REG",regdict[sizeReg])
    tmpload = tmpload.replace("regname","RAX")
    tmpload = tmpload.replace("szereg",str(sizeReg))
    tmpload = tmpload.replace("idreg1",str(1))
    tmpload = tmpload.replace("idfac2",str(MR//2))
    outcode = outcode + "\t" + tmpload + "\n" + "\n"
    regid = 3
    facb = 0
    for nrid in range(NR//2):
        tmpmov = broadcast.replace("REG",regdict[sizeReg])
        tmpmov = tmpmov.replace("regname","RBX")
        tmpmov = tmpmov.replace("idreg1",str(2))
        tmpmov = tmpmov.replace("idfac2",str(facb))
        outcode = outcode + "\t" + tmpmov + "\n"

        tmpmov = broadcast.replace("REG",regdict[sizeReg])
        tmpmov = tmpmov.replace("regname","RBX")
        tmpmov = tmpmov.replace("idreg1",str(3))
        tmpmov = tmpmov.replace("idfac2",str(facb + 1))
        outcode = outcode + "\t" + tmpmov + "\n"
        facb = facb + 2

        regid += 1
        tmpfma = fma.replace("REG",regdict[sizeReg])
        tmpfma = tmpfma.replace("idreg1",str(regid))
        tmpfma = tmpfma.replace("idreg2",str(0))
        tmpfma = tmpfma.replace("idreg3",str(2))
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.

```
            outcode = outcode + "\t" + tmpfma + "\n"

            regid += 1
            tmpfma = fma.replace("REG",regdict[sizeReg])
            tmpfma = tmpfma.replace("idreg1",str(regid))
            tmpfma = tmpfma.replace("idreg2",str(1))
            tmpfma = tmpfma.replace("idreg3",str(2))
            outcode = outcode + "\t" + tmpfma + "\n"

            regid += 1
            tmpfma = fma.replace("REG",regdict[sizeReg])
            tmpfma = tmpfma.replace("idreg1",str(regid))
            tmpfma = tmpfma.replace("idreg2",str(0))
            tmpfma = tmpfma.replace("idreg3",str(3))
            outcode = outcode + "\t" + tmpfma + "\n"

            regid += 1
            tmpfma = fma.replace("REG",regdict[sizeReg])
            tmpfma = tmpfma.replace("idreg1",str(regid))
            tmpfma = tmpfma.replace("idreg2",str(1))
            tmpfma = tmpfma.replace("idreg3",str(3))
            outcode = outcode + "\t" + tmpfma + "\n"

        tmplea = "\n\t" + lea.replace("regname","RAX")
        tmplea = tmplea.replace("jmpfac",str(MR))
        outcode = outcode + "\t" + tmplea + "\n"

        tmplea = lea.replace("regname","RBX")
        tmplea = tmplea.replace("jmpfac",str(NR))
        outcode = outcode + "\t" + tmplea + "\n\n"

        finalcode = finalcode + outcode

    outcode =r""
    idxreg = 4
    for inr in range(NR):
        tmppref = prefetch.replace("regname","RCX")
        tmppref = tmppref.replace("idfac2",str(192))
        outcode = outcode + tmppref + "\n"

        tmpadd = add.replace("regname","RCX")
        tmpadd = tmpadd.replace("REG",regdict[sizeReg])
        tmpadd = tmpadd.replace("idreg1",str(1))
        tmpadd = tmpadd.replace("idreg2",str(idxreg))
        tmpadd = tmpadd.replace("idfac2",str(0))
        outcode = outcode + tmpadd + "\n"
        idxreg = idxreg + 1

        tmpload = loadrtom.replace("REG",regdict[sizeReg])
        tmpload = tmpload.replace("regname","RCX")
        tmpload = tmpload.replace("szereg",str(sizeReg))
        tmpload = tmpload.replace("idreg1",str(1))
        tmpload = tmpload.replace("idfac2",str(0))
        outcode = outcode + tmpload + "\n"

        #tmppref = prefetch.replace("regname","RCX")
        #tmppref = tmppref.replace("idfac2",str(192))
        #outcode = outcode + tmppref + "\n"

        tmpadd = add.replace("regname","RCX")
        tmpadd = tmpadd.replace("REG",regdict[sizeReg])
        tmpadd = tmpadd.replace("idreg1",str(1))
        tmpadd = tmpadd.replace("idreg2",str(idxreg))
        tmpadd = tmpadd.replace("idfac2",str(MR//2))
        outcode = outcode + tmpadd + "\n"
        idxreg = idxreg + 1

        tmpload = loadrtom.replace("REG",regdict[sizeReg])
        tmpload = tmpload.replace("regname","RCX")
        tmpload = tmpload.replace("szereg",str(sizeReg))
        tmpload = tmpload.replace("idreg1",str(1))
        tmpload = tmpload.replace("idfac2",str(MR//2))
        outcode = outcode + tmpload + "\n"

        tmplea = "\n\t" + lea.replace("regname","RCX")
        tmplea = tmplea.replace("jmpfac",str(MR))
        outcode = outcode + "\t" + tmplea + "\n"


    header =r"""
    TEST(RSI, RSI)
    JE(K_LOOP)""" + "\n\t" + "LABEL(LOOP1)\n\n"

    tailer =r"""
    DEC(RSI)
    JNE(LOOP1)
    """ + "\nLABEL(K_LOOP)\n\n"

    finalcode = header + finalcode + tailer + outcode

    # Ending
    reglist = r""
    for i in range(numReg):
        reglist = reglist + "\"" + regdictlc[sizeReg] + str(i) + "\"" + ", "

    return(beginning + initReg(numReg,sizeReg) + setupRegs() + \
           finalcode + ending.replace("reglist",reglist))

print(mainLoop(32,512,2))
```

TREX: Targeting Real Chemical Accuracy at the Exascale project has received funding from the European Union Horizon 2020 research and innovation program under Grant Agreement No. 952165.