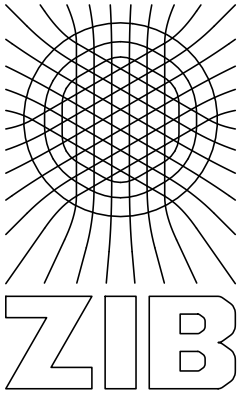
Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany



MATTHIAS NOACK[1], FLORIAN WENDE[1],
GEORG ZITZLSBERGER[2], MICHAEL KLEMM[2] AND
THOMAS STEINKE[1]

[1]ZUSE INSTITUTE BERLIN

[2]INTEL DEUTSCHLAND GMBH

KART
–
A Runtime Compilation Library for Improving HPC Application Performance

ZIB Report 16-48 (October 2016)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

KART – A Runtime Compilation Library for Improving HPC Application Performance

Matthias Noack*, Florian Wende*, Georg Zitzlsberger†, Michael Klemm† and Thomas Steinke*
*Zuse Institute Berlin, Takustraße 7, 14195 Berlin, Germany, Email: {noack, wende, steinke}@zib.de
†Intel Deutschland GmbH, Email: {georg.zitzlsberger, michael.klemm}@intel.com

***Abstract*—The effectiveness of ahead-of-time compiler optimization heavily depends on the amount of available information at compile time. Input-specific information that is only available at runtime cannot be used, although it often determines loop counts, branching predicates and paths, as well as memory-access patterns. It can also be crucial for generating efficient SIMD-vectorized code. This is especially relevant for the many-core architectures paving the way to exascale computing, which are more sensitive to code-optimization. We explore the design-space for using input-specific information at compile-time and present KART, a C++ library solution that allows developers to compile, link, and execute code (e.g., C, C++, Fortran) at application runtime. Besides mere runtime compilation of performance-critical code, KART can be used to instantiate the same code multiple times using different inputs, compilers, and options. Other techniques like auto-tuning and code-generation can be integrated into a KART-enabled application instead of being scripted around it. We evaluate runtimes and compilation costs for different synthetic kernels, and show the effectiveness for two real-world applications, HEOM and a WSM6 proxy.**

I. Introduction

Many C, C++, and Fortran HPC applications are generalized solutions for their respective domains. They typically implement a wide range of algorithms that are meant to be applicable for many different workloads or combinations of input sets. Whilst the applications grow with adding more methods and features there is also a growing demand for applying a subset of methods to restricted workloads. Such use cases do not require the full complexity and flexibility of the original implementations. Hence users strive for optimizing their (limited) workloads by tuning the implementations. This starts with algorithmic optimizations, delivering different implementations of the original algorithm, such as reducing dimensions or replacing algorithms that work better for smaller problem sizes. To give a few examples, for the WSM6 proxy code [1], the authors provided constants at compile time and achieved a more than 40% performance gain on the Intel® Xeon Phi™ coprocessor (former codename Knights Corner, KNC). For the DL_MESO code it was recently demonstrated [2] that if increasingly more source code constants are known at compile time a performance improvement of more than five times can be achieved by enabling better SIMD optimizations for the compiler's auto-vectorizer.

At some point in this process, implementations in source code are specific enough to be further optimized by compilers. Compilers typically do not apply algorithmic optimizations that change the semantics, but they have a rich set of optimizations to enhance code generation. However, compiler optimizations for C, C++, and Fortran can only be applied upon the provided algorithmic implementations and need to be general and adhere to language standards. Knowledge about the runtime context of a unit of code would allow to optimize for specific memory access strides, eliminate conditional code, or apply workload-dependent loop transformations.

A typical approach to remedy this is to apply multi-versioning, that is, generating multiple specialized instantiations of the same function, loop, or code fragment. This can be achieved by a programmer using dedicated implementations, like C++ template specializations, or preprocessor macros, for example. Compilers can also emit versioned code to handle aligned versus unaligned data, to create different code paths for different instructions sets (e.g., Streaming SIMD Extensions and Advanced Vector Extensions), or to avoid SIMD vectorization for too small loop trip counts, to just name a few. Because multi-versioning can dramatically increase the code size, compilers usually only generate a few code versions and provide a general fallback code path. For the large combinatorial space spanned by the potential inputs of an HPC application, multi-versioning becomes ineffective.

Programmers can try to help the compiler by adding compilation hints (e.g., pragmas/directives or attributes) to limit the amount of code versions. But even if a programmer provides different implementations there are limits. Optimizations can only be applied for a small set of categories of workloads, and also lead to code size increase which can make an implementation harder to maintain. While it is quite simple to provide different optimizations for different dimensionality of input data sets, it is much harder to do so for different memory access patterns, access strides, or loop trip counts. There are far too many different goals to optimize for, and grouping them into categories for directed optimization is hard.

The main contributions of this work are the exploration of the design space for exploiting runtime data for compiler optimization, a light-weight, flexible runtime-compilation framework (KART), and its evaluation. Our solution is to recompile algorithms (kernels) during the runtime of an application, thereby optimizing within the current context of kernel execution. This especially allows to optimize for values that manifest as constants during runtime but were not known at compile time. KART is general enough to benefit a wide range of applications without being limited to a certain back-end. The approach is particularly relevant for many-core architectures

like the Intel Xeon Phi (co)processor, whose microarchitecture is more sensitive to code-optimizations.

For frequently used kernels, we show that the improved optimization outweighs the runtime compilation overhead. We also regard this approach as a solution for OpenMP [3] applications to benefit from the same dynamic recompilation advantages that OpenCL™ [4] and CUDA [5] provide.

II. Related Work

In the following section we will exclude work which is focused on just-in-time compilation for interpreter or scripting languages (e.g., Python, Perl, Lua). We are aware that Java and Microsoft's .NET Framework rely on JIT compilation for high-speed code execution, but both have still a small representation in the HPC domain.

The LLVM compiler infrastructure [6] includes a JIT code generation system which supports various CPU architectures, and provides a foundation for just-in-time compilation projects. For the QCD application domain, the QDP-JIT/LLVM approach [7], which extends the ideas of QDP-JIT/PTX [8], uses C++ expression templates to implement LLVM IR code generators that emit executable code via the LLVM JIT component. In Julia, the language design is combined with the high-performance LLVM-based JIT compiler [9], [10]. This enables code generation that comes close to the performance of a C implementation.

The LIBXSMM library [11] for small dense matrix-matrix multiplications enables static and JIT code generation for Intel® Xeon® processors and Intel Xeon Phi (co)processors. While the basic ideas of LIBXSMM are similar to our proposal in that it specifically compiles kernels for a particular target architecture, it only supports *sgemm* and *dgemm* with restricted *alpha* and *beta* inputs.

In the context of MPI communication, Schneider et. al. [12] demonstrate that through runtime compilation of (un)pack functions for non-contiguous data in MPI an order of magnitude better performance can be achieved over the interpretation scheme found in today's MPI implementations.

In contrast to the specialized solutions above, our approach is more general, as it applies to arbitrary code and allows use of different C, C++, and Fortran compilers.

III. Design Space

A. Recompile Everything

The most straight-forward approach to using input-specific data as compile-time constants would be to simply recompile the whole application for each set of input data. It completely avoids the complexity of additional compilation at runtime and enables the compiler to apply cross-module optimization techniques like whole-program optimization. However, large parts of the code are typically not performance critical and recompiling them would introduce a prohibitive compilation overhead that grows with the size of the code base and optimization levels. For large codes with compile times in the range of hours, this is a significant impact on time-to-solution and renders quality assurance procedures less feasible.

An important question for this approach is: How to acquire the needed data from the input at build time? Typically, reading and parsing of input data is done at runtime during the initialization phase. So some application code has to be executed to get the data required for building the application. To break this circular dependency, an input parser would be required to analyze the input data and to provide the necessary constant values before the build process can be started.

A concern for some developers, e.g., of commercial codes, might also be that binary releases of applications (or libraries) would still require to contain the runtime-compiled code sections as source.

B. Pre-instantiate Code for all Cases

A different approach would be to prepare for a potential and classified set of inputs. Performance critical functions would be instantiated or specialized multiple times with different classes of compile-time constants when compiling the application. This is similar to what some compilers already do when generating multiple versions of functions or loops, e.g. to provide both a scalar and a vectorized version. Such multi-versioning is bound by combinatorial effects of the parameter space and for all input variables their numerical domain needs to be known at compile time.

One manual implementation approach is to use C++ templates with desired constants as template parameters, i.e. classification of values. Those can be explicitly instantiated to collect the resulting function pointers in a map with their classification as the key. Additionally, a fall-back implementation without compile-time constants but a set of additional parameters could be provided. At runtime, the calling code uses actual values from the input to select the appropriate function template specialization, or a fall-back if no suitable instantiation was found. For dimensions of domains this might be applicable as those are typically limited. However, optimizing for different sizes becomes infeasible if their values cannot be restricted to a small amount of classes.

C. Call a Compiler Library at Runtime

Another solution that is already commonly used and inspired this work is OpenCL that aims for portability across heterogeneous compute devices, such as CPUs, GPGPUs, Intel Xeon Phi (co)processors, or FPGAs. It divides the application into host code for a host processor and device code that runs on a compute device and is compiled at runtime. This enables portability, but can also be used for passing runtime-data from the host program into the compilation of the device code. Previous work has shown that there can be a performance benefit when compile-time constants are known to the OpenCL compiler [13]. Since recently, Nvidia's CUDA GPGPU framework provides similar means [14].

Porting an existing (HPC) application to OpenCL is a major effort and requires rewriting code using OpenCL's kernel language, explicitly managing kernel invocations and memory transfers between host and compute device.

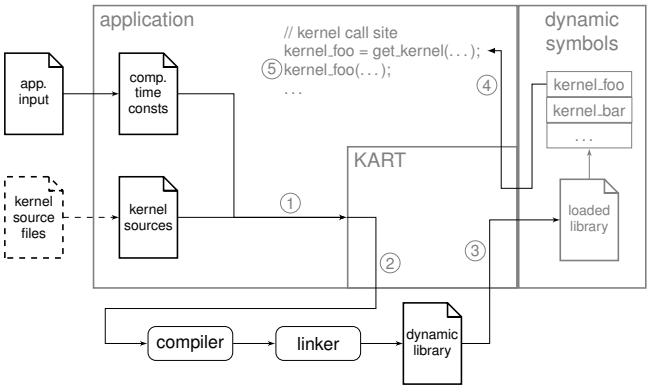


Fig. 1. Schematic of KART: 1) Compile-time constants derived from the input, and kernel source code are passed to KART. 2) KART starts a system compiler and linker to create a dynamically linked library. 3) The library is dynamically linked into the application. 4) The application queries KART using the kernel's function name to get a callable function pointer. 5) The kernel is invoked.

OpenMP supports heterogeneity but is limited to one code version and does not provide runtime compilation. C, C++, or Fortran compilers do not expose a library API that could be used in an OpenCL-like fashion. LLVM with its modular architecture would be a viable basis to implement such a mechanism. The major drawback of an LLVM-based solution is the limitation to mere LLVM optimization abilities. OpenMP support of LLVM is still not complete, although catching up fast [15], [16].

D. Call Arbitrary Compilers at Runtime

A more abstract and flexible solution is to provide an API to use any installed command-line compiler from within the application and then incorporate the resulting object code into the running program. Being close to the OpenCL model, it yields the highest flexibility, as any compiler, even for different languages, can be used. It can defer the compilation of performance critical-code sections until execution time and apply the best-optimizing compiler for a specific code fragment and target architecture, or to multi-version a kernel using different compilers and apply auto-tuning. Also different (hand-written) implementations of the same kernel can be used. As such, optimization is not only defined by compiler capabilities and varying compile-time constants, but also by the user at algorithmic level. Kernels can be supplied in languages different from the one chosen for the host application, which increases the flexibility, e.g., by using C SIMD intrinsics within a Fortran application. This is the approach we have chosen for the solution presented in this paper.

IV. The KART Library

A. Design and Implementation

KART provides application developers with the means to compile and use pieces of code at runtime with a minimal amount of overhead and maximal flexibility. KART resembles a minimal build system with a library interface.

KART offers a slim API to compile a code fragment given either as a text string or a source file, and to call the result after compiling the code. KART is implemented in C++ using several Boost libraries [17] and provides APIs for C, C++, and Fortran, so far.

The intended flexibility to use any compiler requires the invocation of command-line compilers and linkers in separate processes at runtime to create a shared library from a given source code. Subsequently, the resulting library is linked to the running program via `dlopen()`, and the contained functions can be accessed and executed using their (symbolic) name. Compile time constants can be integrated into the runtime-compiled code by either generating the corresponding lines before passing the code to KART, or can be specified using compiler command line options (e.g., `-DNAME=VALUE`) for which KART provides an interface, too. Figure 1 illustrates the basic working principle of KART.

Figure 2 shows a class diagram containing the C++ API of KART together with its abstractions. The API provides a *toolset* abstraction that encapsulates the specification of a compiler and linker command, as well as different sets of options for these commands. Toolsets are defined in small configuration files. A default toolset file can be set via an environment variable. Once a toolset object is created from a configuration file, additional options can be specified, e.g., to add constant definitions.

The second main abstraction is the *program*, which is constructed from either a string or a file containing the source code. The program is then built using the selected toolset and once built, callable function pointers can be acquired by specifying a name, and optionally a signature to ensure type-safety in C++. The application must know and use the correct signature of the called kernel.

Already built programs can be rebuild with a different toolset. This is more efficient than creating new program objects from the same source. The motivations behind this functionality are auto-tuning and benchmark codes that apply many configurations to the same code. Another use-case is applications with temporary run-time constants, for instance loop trip counts or sizes of data structures that may change after a load-(re)balancing step between compute nodes.

A rebuild invalidates existing pointers, which is handled by using smart function pointers (called *kernel_ptr*). These pointers are callable objects that are transparently used like a function with negligible overhead. Each kernel_ptr is registered with its corresponding program on construction, and unregistered on destruction. This allows the program object to update all pointers after a rebuild has been performed. When a program goes out of scope, all registered objects are invalidated to prevent scoping errors.

In order to cover the most relevant languages for HPC applications, there is also an API for C and Fortran. The C API is a wrapper around the C++ implementation that uses opaque handles and functions, instead of objects and methods. The Fortran interface is a set of bindings for the C API.

Internally, KART uses the POSIX calls `dlopen()`,

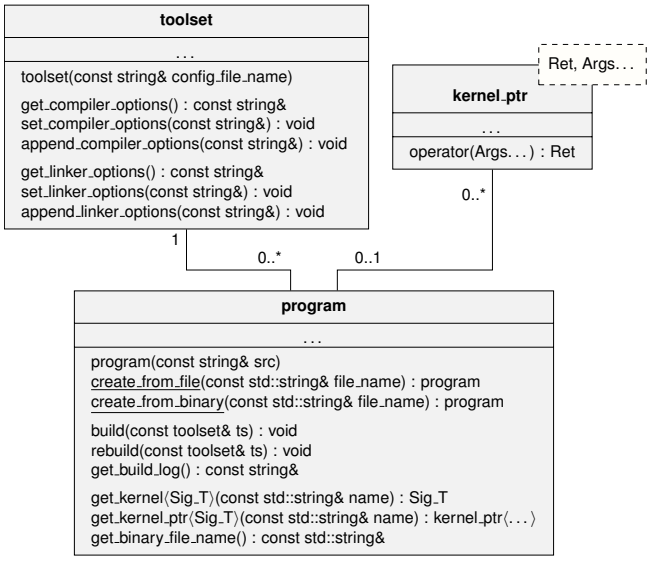


Fig. 2. This simplified class diagram depicts the main abstraction of KART, their relation, as well as the basic C++ API. *toolset* class encapsulates the compiler and linker command line tools, and their corresponding options. The *program* class is used to build source code using a toolset, or from an already built library. Callable kernel objects can either be acquired as raw function pointers, or safer *kernel_ptr* objects.

```
#include "kart/kart.hpp"

// signature type
using my_kernel_t = double(*)(double, double);
const char my_kernel_src[] = R"kart_src(
extern "C" {

// original function
double my_kernel(double a, double b)
{ return a * b * CONST; }

})kart_src"; // close raw string literal

int main(int argc, char** argv)
{
  // create program
  kart::program my_prog(my_kernel_src);
  // create default toolset
  kart::toolset ts;
  // append a constant definiton (runtime value)
  ts.append_compiler_options(" -DCONST=5.0");
  // build program using toolset
  my_prog.build(ts)
  // get the kernel
  auto my_kernel =
       my_prog.get_kernel<my_kernel_t>("my_kernel");

  /* ... application code ... */

  // call the kernel as usual
  double res = my_kernel(3.0, 5.0);

  /* ... application code ...  */
}
```

Fig. 3. This example shows how to embed a kernel as source code, and compile it at runtime using KART. The highlighted lines show what is needed to introduce KART. Raw string literals, as provided by C++11 can be used to embed source code without having to escape some characters.

`dlsym()`, and `dlclose()` for interfacing with the generated shared libraries. This allows the use of different compilers and even languages within the same application—as long as the resulting libraries are binary compatible. The application, as well as the runtime-compile code can be still statically linked. There are some ABI-based constraints, especially when combining compilers of different major versions, but we have not found any issues when combining the typical GNU, Intel, and LLVM/clang compilers for C, C++, and Fortran. Whereas C and C++ have a fixed ABI on Linux, it can be problematic to mix different Fortran compilers. It is recommended to use the Fortran ISO-C bindings to have a common ABI.

GNU-compiled C/C++ applications can use the most recent Intel® C++ Compiler to generate performance-critical code using its vectorization capabilities in addition to the input-specific compile-time constants. Fortran codes could use a C/C++ compiler to facilitate manual vectorization for kernels via C SIMD intrinsics, for instance. In addition, a benchmark or auto-tuning code could automatically evaluate different compilers and sets of options, e.g., for different optimization levels, pre-fetching settings, or numerical precision levels.

Paths and names for temporary files can be configured using environment variables or the API, respectively. The internally generated library binaries can be acquired from the program object, and a program object can be initialized by a pre-built shared library. This provides the flexibility needed to deal with various application architectures found in the HPC context, for instance, coprocessor offloading, message passing, or complex tool pipelines.

B. Usage

KART was designed with ease of use in mind. Figure 3 shows a simple example, where the function `my_kernel` is compiled at runtime. For a single function, wrapping the original code into a raw string literal, as shown in the example above, is sufficient. Using an `extern "C"` block makes sure the kernel's name does not get mangled by the compiler and can be used directly. However, if the needed source code already is a separate compilation unit or gets larger, it is more convenient to use source files instead of embedded strings.

There are two ways to specify dependencies and other compile/link options: toolset configuration-files, intended for the compiler and the host-specific part (often non-portable), and methods called at runtime for the application-specific part. This way, the source code remains portable.

For existing code, the most convenient way of integrating runtime compilation would be a directive-based approach, where code is simply annotated as runtime compilation target. A mechanism like the one provided by KART could become part of a widely accepted and standardized programming model like OpenMP in a future version. However, this would mean giving up the flexibility to use any compiler and the library-only implementation.

1) Adapting existing code: When adapting a code, the best method is identifying hotspots whose index computations, memory access patterns, loop counts, and branching predicates depend on input data. Once identified, it can be recompiled using compile-time constants for a few inputs to estimate the

potential gain before restructuring the code. The runtime gain determines an upper bound for the acceptable compilation overhead. The process is very similar to adapting an application for offloading to an accelerator—without the need to rewrite kernels in another language and optimize them for the accelerator's architecture.

The intrusiveness of incorporating runtime kernel compilation into an existing code base depends on the current code structure, as the build-time and run-time compiled source needs to be separated. For a well-structured code base, this means identifying the compilation units and adding the KART API calls into the application's initialization phase.

Most build systems provide a verbose mode that prints out the compile and link commands used. That is the natural starting point to generate a toolset specification for KART that include necessary flags and dependencies. In a second step, the dependencies can be minimized, and compile flags further optimized to improve compilation and runtime of the kernel, respectively.

The following sections describe how KART can be used in different HPC application patterns.

2) Coprocessors: Applications for the Intel Xeon Phi coprocessor often use an offload programming model like OpenMP or Intel's LEO (Language Extensions for Offload). In such a setting, KART can be called prior to the offload to cross-compile the coprocessor code leveraging the better single-thread performance of the host CPU. For the first-generation of Intel Xeon Phi coprocessor (Knights Corner), cross-compilation is even mandatory, as there is no highly optimizing compiler available running natively on the coprocessor.

KART allows the developer to specify the directory and file name for the created binary, which in turn allows using a shared file system for direct access. The offload only needs the path to the binary as an argument, from which a KART program object can be created inside the offloaded code. Within a more flexible offloading framework, like HAM Offload [18], a reverse offload from a native coprocessor application to the host can be used to offload the compilation, even remotely over a fabric.

Another possibility would be a symmetric MPI job with processes on both, the coprocessor and the host. The host rank(s) could then do the compilation work for the coprocessors in a similar fashion as described above, communicating either the path on a shared filesystem, or the binary itself.

3) OpenMP: OpenMP can be used within KART-compiled kernels or higher up in the calling tree. We have successfully tested both. Code that is built at runtime using KART is and behaves like a shared library. So as with other libraries that make use of OpenMP, the developer has to make sure that there is no conflict between the OpenMP implementation used in the application code and the one used inside the library. Other than that, there are no known limitations.

4) MPI: For large distributed jobs, solely using KART on the node level is the easiest way, but not always the best. It is beneficial in cases where every node uses different constants, for example if loop counts for local partitions of an irregular grid are used. In cases where every node does the same work, there is potential for optimization. The situation can be used for an auto-tuning step, where different compilation options or kernel variants are built and timed, followed by an exchange of the timings and the best-performing kernels. Ranks can exchange their compilation results via a distributed file system or by transferring the generated binaries as messages. If that is not applicable, factoring out the compilation step into a small prologue job that runs on a single node, processes the input, and pre-builds the kernel binaries for the big job might be an option. KART can also use wrapper compilers if the runtime-compiled code uses MPI directly.

A less technical but more legal problem is the licensing issue. On large clusters and supercomputers, commercial compilers are often using a license server with a limited amount of licenses. Depending on the configuration, there is a chance for an unintended "denial-of-service attack" against a license server, by taking all licenses of a shared system hostage. So it is either impossible, or at least not desirable, to check out thousands of licenses. For these cases, compilation needs to be limited to a few ranks, or be performed using the prologue-strategy.

5) Auto-tuning: KART can be used to easily implement auto-tuning and also to complement it orthogonally. Within an applications, the use-cases are slightly different: KART can address given runtime-values, not subject to tuning, by making them a compile-time known value enabling better optimization. For instance, given a tunable value like a block-size, KART could be used to auto-tune through a range of values, e.g., distributed across ranks of an MPI-job during initialization, while using the input-specific overall size of the blocked data-structure as a compile-time constant. Where auto-tuning typically generates multiple versions for a wide variety of options/inputs with later run-time selection, KART generates the code for the current scenario at runtime.

Beside the described HPC-relevant patterns and use cases, the availability of a general runtime compilation mechanism can be exploited in more sophisticated ways. There is no fixed pattern. For instance, instead of just using compile-time constants or different compilers, languages, and options for existing code, KART can be used as a back-end for code generators and domain-specific languages. In contrast to existing specialized solutions (see Section II), KART provides a maximum of flexibility for a broad variety of use-cases.

V. Evaluation

A. Benchmarks

To demonstrate the motivation for KART, we exemplify the potential of runtime kernel compilation with two simple and synthetic benchmarks. We also demonstrate the achieved speed-ups for two tested real world applications. Table I lists the hardware used for benchmarking. The used software versions are: Intel OpenCL Runtime 16.1.1, Intel C++ Compiler 16.0.3, GCC 6.1.0, and Clang 3.8.1.

TABLE I
CHARACTERISTICS OF THE PROCESSOR PLATFORMS USED FOR BENCHMARKING. THE XEON SYSTEM IS A DUAL SOCKET NODE.

	Intel Xeon processor E5-2630 v3 (HSW)	Intel Xeon Phi processor 7210 (KNL)
Cores	8	64
Threads	16 (8×2)	256 (64×4)
SIMD Width	256 Bit	512 Bit
Frequency	2.4 GHz	1.3 GHz
MCDRAM	—	16 GiB (ECC)
Memory BW	59 GiB/s	102 / 352 GiB/s
TDP	85 W	215 W

1) Runtime Compilation: On principle, runtime compilation always introduces overhead. Hence, it is important to understand the trade-off between runtime compilation overhead and kernel speed-up first. The overhead introduced by KART is largely determined by the runtime of the invoked compiler and linker. The speed-up of the runtime-compiled kernel without incorporating the compile time over the reference kernel is $s_b = t_{\text{ref}}/t_{\text{kart}}$, with $s_b > 1$ for $t_{\text{ref}} > t_{\text{kart}}$. It is an upper bound for the actual speed-up that includes the compile time, which is $s = \frac{n \cdot t_{\text{ref}}}{n \cdot t_{\text{kart}} + t_{\text{compile}}}$ with n being the number of kernel calls. Numerator and denominator are two linear functions for the respective overall runtime cost of the reference and KART-compiled kernel. For given run and compile times, there is an n_c where both functions cross and $s = 1$. For every $n > n_c$, there is an actual application speed-up, asymptotically approaching s_b. Runtime compilation techniques pay off when the accumulated runtime savings of all kernel calls exceed the runtime compilation cost.

Figure 4 shows the compilation cost using OpenCL as a reference and KART with different compilers. These are roughly the same timings as measured when executing the command lines generated by KART by manually typing them into a console terminal—the cost of the KART API itself is negligible. The linking step took roughly two thirds of the time. The timings for the empty kernels show that there is a large constant cost coming with the command line compilers. This includes starting processes and lots of file operations when handling dependencies like a large set of header and library files, all not present in OpenCL. A library interface to existing compilers together with a set of small headers and libraries specifically optimized for compilation time could improve the situation. For commercial compilers, additional time is lost for fetching licenses from a file system or network. Caching the compilation results between application runs could mitigate these costs if the actual reuse is high.

For all use-cases that only need a single runtime build per kernel during initialization, a few seconds are tolerable, given that the kernel runtimes for computational hotspots can easily add up to many minutes or even hours during large production runs. For more dynamic use cases, where kernels are regularly rebuilt as values of compile-time constants need to be adapted, we recommend profiling the performance gain

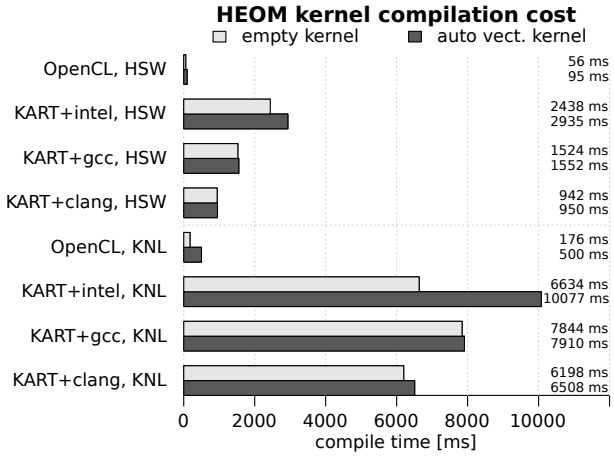


Fig. 4. Comparison of runtime compilation costs using OpenCL and KART with different compilers for the OpenMP-vectorized Hexciton kernel of the HEOM benchmark, and an empty kernel. There is a high constant cost for any compilation regardless of the kernel size. Only the Intel C++ Compiler adds a significant cost to the empty kernel, but also generates the fastest code in case of auto vectorization. OpenCL's compiler with a library ABI and also less indirectly compiled code from standard library headers is two orders of magnitude faster. The Xeon Phi processor (KNL) compilation times suffer from the lower single thread performance compared with a Xeon (HSW).

per kernel invocation, the compilation cost, and the frequency of recompilation. The large constant cost of each compiler invocation can be distributed among multiple kernels by using a single program object that aggregates all sources.

2) Synthetic Kernels: Demonstrating the potential of runtime kernel compilation highly depends on the kernel itself and the context it is invoked in. It also depends on the used compiler, compiler options, and processor architecture. We have selected two simple kernels to highlight the principal usefulness of runtime compilation for HPC kernels:

- *convolve*: One-dimensional linear convolution with an offset into the input vector *input*. This offset could be seen as a way to decompose the input and assign subsets to different threads. Figure 6 shows the KART version.
- *matvec*: Matrix vector multiplication with scaling (*alpha*). To give the compiler's optimizer some optimization headroom, we assume the scaling to be 0.0 or 1.0, and matrix *a* to have just one column. This simulates cases where the compiler can remove invariant statements and/or loops. Compared to *convolve*, which is only called once, the *matvec* kernel is executed multiple times. Figure 5 shows the KART version.

Both kernels have been compiled via KART (as seen in the listing) and as ordinary C functions with all parameters as arguments (e.g., *alpha*, *rows*, and *cols* for kernel *matvec*). KART was used to pass the static values via preprocessor macros (`-D...`).

The runtimes for the Intel C++ compiler are shown in Figure 7 (the compiler options used were always identical: `-restrict -std=c++11 -qopenmp -xHost -O3`).

For the *matvec* kernel the compiler was able to entirely eliminate the loops for *alpha=0* and even showed a 3.0×

```
extern "C"
void matvec_kart(float a[][COLS],
                 float b[ROWS],
                 float x[COLS])
{
  for (int i = 0; i < ROWS; ++i)
    for (int j = 0; j < COLS; ++j)
      b[i] += a[i][j] * x[j] * ALPHA;
}
```

Fig. 5. The synthetic *matvec* kernel, a benchmark for matrix vector multiplications. `COLS`, `ROWS`, and `ALPHA` are the introduced compile-time constants.

```
extern "C"
void convolve_kart(float* restrict input,
                   float* restrict kernel,
                   float* restrict output)
{
  #pragma omp parallel for
  for (int i = 0; i < INPUT_SIZE; ++i) {
    float sum = 0;
    for (int j = 0; j < KERNEL_SIZE; ++j)
      sum += kernel[j] * input[OFF + i + j];
    output[i] = sum;
  }
}
```

Fig. 6. The synthetic *convolve* kernel, implementing a one dimensional convolution algorithm. `INPUT_SIZE`, `KERNEL_SIZE`, and `OFF` are the introduced compile-time constants.

kernel speed-up for *alpha=1* where the inner-most loop was removed since there is only one column of matrix *a*. For the *convolve* kernel, having *off*, *input_size*, and *kernel_size* known by the compiler could improve kernel runtimes by 16.9×.

Those cases are just examples to show the potential that can be unleashed through runtime code generation and optimization. Especially, if the input data has certain properties that can be exploited by compiler optimizations of a kernel, the effects can be quite high.

B. Real World Applications

1) HEOM: HEOM is a short for Hierarchical Equations of Motion, which is a mathematical approach to solving open quantum systems. It is used, for instance, to simulate the energy transfer in photo-active molecular complexes to understand recent observations of femto-second laser-experiments on photosynthesis molecules found in bacteria and plants.

For the GPU HEOM OpenCL implementation [19], a detailed case study [20] including a benchmark is publicly available. It compares a variety of OpenCL and OpenMP implementations of the Hexciton kernel as well as different vectorization and optimization strategies. We integrated KART into the OpenMP version of the benchmark to build the different kernel variants at runtime and enable the use of input-specific constants at kernel compile time. Two constants, the matrix dimension and the number of matrices of the central HEOM data-structure, are most relevant for the memory access pattern and the loop counts. Figure 8 shows the achieved performance gain. The plotted values are from the set of the best-performing kernel variants using the Intel C++ Compiler. The highest speed-up of 2.6× was observed for clang++ on the Xeon CPU with the manually vectorized kernel, but the absolute runtime was still slower than that of the code emitted by the Intel C++ Compiler.

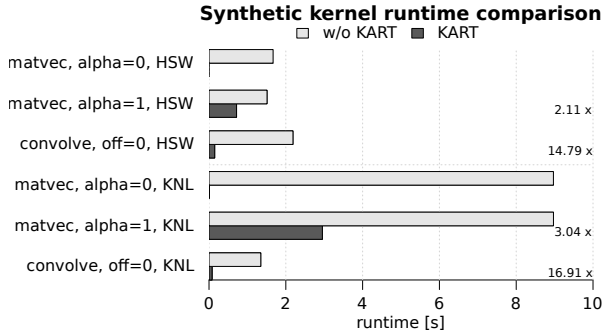


Fig. 7. Speed-ups for two synthetic kernels *matvec* (single threaded) and *convolve* (OpenMP). The speed-ups do not include the compilation overhead, as it this would require defining a somehow "realistic" number of kernel calls. A compile-time known *alpha=0* entirely removed the computation loops. The average compilation times for each kernel ranges from 0.90 to 0.92 s on the Xeon (HSW) and 3.46 to 3.60 s on the Xeon Phi processor (KNL).

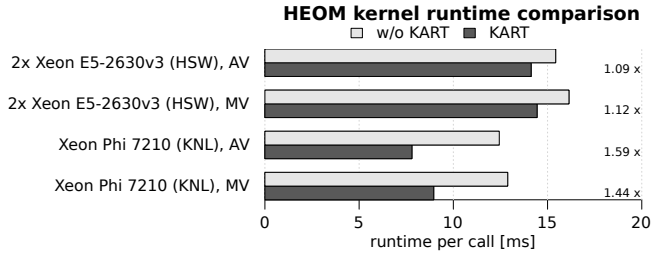


Fig. 8. Runtimes and speed-ups achieved by using KART for the OpenMP version of the HEOM benchmark. Matrix size and number of matrices read from the input are used as compile-time constants during runtime compilation. AV is automatic and MV is manual vectorization. Amortization of compile time requires 659 and 7195 calls, 90 % of the theoretical gain ($s_b - 1$) are reached after 7195 and 24431 calls, on Xeon (HSW) and Xeon Phi (KNL), respectively. Typical application runs need 4000 to 40000 calls.

The Intel Xeon Phi (co)processor benefits much more from the compiler optimizations. Its light-weight cores and the missing L3 cache requires better optimized code than the Haswell architecture. The effect on the whole HEOM application runtime cannot be measured as only the Hexciton kernel has been ported to OpenMP. However, it is the computationally most complex part of the solver, and the remaining computations will most likely benefit in a similar way as they work on the same data structure using the same constants. The compile-time overhead is not relevant for production runs of the HEOM code, as they typically take several hours.

2) WSM6 Proxy Code: We integrated KART into the WSM6 proxy code [1] written in Fortran for demonstrating different optimization techniques. WSM6—the WRF SIngle Moment 6-class Microphysics schema—is part of the Weather Research and Forecast (WRF) model, widely used for numerical weather prediction. The authors investigated the impact of known compile-time constants describing loop lengths and array dimensions on the efficiency of the generated code.

The original benchmark uses the C preprocessor and a set of Perl scripts to modify the source code to generate a version

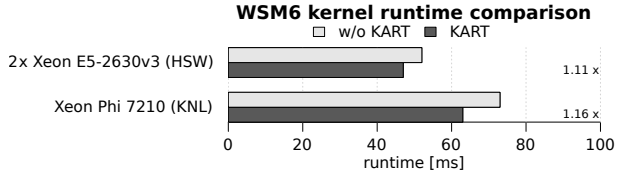


Fig. 9. Runtime comparison and speed-ups for using runtime compilation with the WSM6 Kernel on the Xeon (HSW) and Xeon Phi (KNL) architecture. Average compilation times are 3.24 s on the Xeon (HSW) and 20.13 s on the Xeon Phi processor (KNL).

with compile time constants for every run. This is no longer necessary when using KART. Our modified Kernel uses the preprocessor only to simplify the build process at runtime. For the WSM6 proxy code, KART achieved a speed-up of 1.16× for the slightly modified kernel (see Figure 9).

VI. Conclusion and Outlook

Studies on the optimization of HPC workloads for many-core CPUs demonstrate the impact of available constants on the optimization capabilities during the compilation step. In particular, the more parameters affecting the data layouts and loop/branching structures can be provided, the better the compiler is able to optimize. Compiling hot-spot kernels at run-time for the context of their invocation can yield better optimizations and thus shorter time-to-solutions.

We have presented KART, a flexible and easy to use framework that allows runtime compilation with a variety of C, C++, and Fortran command line compilers. It supports dynamic re-compilation during program execution if kernel parameters such as data sizes are changed, e.g., after load-balancing steps between nodes or to adjust for different input data sets. For applications with irregularly distributed data across the compute nodes, our approach can support individually optimized kernels for each node.

We have demonstrated the effectiveness of our solution for synthetic and real-world kernels. Potential speed-ups can be significant, depending on the specifics of the kernel and target architecture. KART considerably accelerated the HEOM (1.59×) and WSM6 (1.16×) kernels with an acceptable compilation overhead, given the typical runtimes of HPC applications. The approach is particularly effective on the Intel Xeon Phi (co)processor, which represents a many-core architecture whose successors are a likely technology for exascale systems. The convenience and efficiency of runtime compilation could be further improved if it would be integrated into a programming model like OpenMP or if compiler vendors would provide an—ideally standardized—API to their tools.

KART and the benchmarks are available as open source on GitHub (https://github.com/noma/kart). We expect further improvements of KART and adoptions by other HPC applications in the near future and are currently working on automatic caching mechanisms for compiled kernels.

Acknowledgments

This work is partially supported by Intel Corporation within the "Research Center for Many-core High-Performance Computing" (Intel PCC) at ZIB. We thank the "The North-German Supercomputing Alliance - HLRN" for providing us access to the HLRN-III production system 'Konrad' and the Cray TDS system with Intel KNL nodes.

References

[1] T. Henderson, J. Michalakes, I. Gokhale, and A. Jha, "Chapter 2 - Numerical Weather Prediction Optimization," in *High Performance Parallelism Pearls*, J. Reinders and J. Jeffers, Eds. Boston: Morgan Kaufmann, 2015, pp. 7 – 23.

[2] S. Siso, "DL_MESO Code Modernization." Intel Xeon Phi Users Group (IXPUG), March 2016, IXPUG Workshop, Ostrava.

[3] OpenMP Architecture Review Board, "OpenMP Application Program Interface, Version 4.5," 2015, http://www.openmp.org/.

[4] Khronos OpenCL Working Group, "The OpenCL Specification, Version 2.2," March 2016, https://www.khronos.org/registry/cl/specs/opencl-2.2.pdf.

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[6] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *CGO*, San Jose, CA, USA, March 2004, pp. 75–88, llvm.org.

[7] B. Joó, "LLVM and QDP-JIT," https://www.ixpug.org/events/ixpug-annual-meeting-2015, September 2015, iXPUG Workshop, 2015, Berkeley.

[8] F. T. Winter, M. A. Clark, R. G. Edwards, and B. Joó, "A Framework for Lattice QCD Calculations on GPUs," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1073–1082. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.112

[9] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," September 2012, http://julialang.org.

[10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," November 2014.

[11] A. Heinecke, H. Pabst, and G. Henry, "LIBXSMM: A High Performance Library for Small Matrix Multiplications," http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post137.html, November 2015, poster.

[12] T. Schneider, F. Kjolstad, and T. Hoefler, "MPI Datatype Processing using Runtime Compilation," in *Proceedings of the 20th European MPI Users' Group Meeting*. ACM, Sep. 2013, pp. 19–24.

[13] A. Heinecke, M. Klemm, D. Pflüger, A. Bode, and H.-J. Bungartz, "Extending a Highly Parallel Data Mining Algorithm to the Intel® Many Integrated Core Architecture," in *Euro-Par 2011: Parallel Processing Workshops*, Bordeaux, France, August 2011, pp. 375–384, lNCS 7156.

[14] NVIDIA, "NVRTC - CUDA Runtime Compilation User Guide," September 2015, http://docs.nvidia.com/cuda/pdf/NVRTC_User_Guide.pdf.

[15] "OpenMP: Support for the OpenMP language," http://openmp.llvm.org/, April 2016.

[16] "OpenMP Compilers," http://openmp.org/wp/openmp-compilers/, September 2016.

[17] B. Schling, *The Boost C++ Libraries*. XML Press, 2011.

[18] M. Noack, F. Wende, T. Steinke, and F. Cordes, "A Unified Programming Model for Intra- and Inter-Node Offloading on Xeon Phi Clusters," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, 2014, pp. 203–214. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.22

[19] C. Kreisbeck, T. Kramer, and A. Aspuru-Guzik, "Scalable High-Performance Algorithm for the Simulation of Exciton Dynamics. Application to the Light-Harvesting Complex II in the Presence of Resonant Vibrational Modes," *Journal of Chemical Theory and Computation*, vol. 10, no. 9, pp. 4045–4054, 2014, pMID: 26588548. [Online]. Available: http://dx.doi.org/10.1021/ct500629s

[20] M. Noack, F. Wende, and K.-D. Oertel, "Chapter 19 - OpenCL: There and Back Again," in *High Performance Parallelism Pearls*, J. Reinders and J. Jeffers, Eds. Boston: Morgan Kaufmann, 2015, pp. 355 – 378.