KART – Kernel compilation At RunTime for Improving HPC Application Performance

Matthias Noack (noack@zib.de), Florian Wende, Georg Zitzlsberger, Michael Klemm, Thomas Steinke

Zuse Institute Berlin
Distributed Algorithms and Supercomputing
Problem

Information that could dramatically improve compiler optimisation, i.e. application runtime, is not available at compile-time.
Motivation

Real-World Example ...

- ...from porting an OpenCL kernel to OpenMP
- SIMD vectorisation ⇒ AoSoA memory layout ⇒ complex index computations

```c
// in a loop nest: group_id (parallel), local_id (simd), i, j, k
sigma_out[group_id * VEC_LENGTH * 2 * DIM * DIM + 2 * VEC_LENGTH * (DIM * i + j) + local_id]
```

- without the input runtime-constant DIM the compiler does not recognise the contiguous memory accesses pattern ⇒ gather/scatter SIMD loads/stores
- defining DIM at compile-time yields contiguous loads/stores ⇒ up to 2.6x
Problem

Information that could dramatically improve compiler optimisation, i.e. application runtime, is not available at compile-time.

- dependant on runtime constants
  - e.g. input data, number of nodes in a job, partitioning, data layouts, etc.
  - conditional elimination, loop transformation, memory access optimisation, ...
  - enable/improve SIMD vectorisation

Solutions?

a) at application compile time
  - recompile code for a specific runtime scenario (input)
  - pre-generate code versions for a possible parameter space

b) defer compilation of kernels (i.e. hotspots) until application runtime
  - OpenCL does that by design (for hardware portability)
  - CUDA since recently via NVRTC extension
  - OpenMP (and others) cannot
Problem

Information that could dramatically improve compiler optimisation, i.e. application runtime, is not available at compile-time.

- dependant on **runtime constants**
  - e.g. input data, number of nodes in a job, partitioning, data layouts, etc.
  - conditional elimination, loop transformation, memory access optimisation, ...  
  - **enable/improve SIMD vectorisation**

Solutions?

a) at application compile time
   - recompile code for a specific runtime scenario (input)
   - pre-generate code versions for a possible parameter space

b) **defer compilation of kernels (i.e. hotspots) until application runtime**
   - OpenCL does that by design (for hardware portability)
   - CUDA since recently via NVRTC extension
   - **OpenMP (and others) cannot**
A. Recompile Everything

- process input somehow at build time
- use data for compilation

✓ no runtime compilation complexity
✓ cross module optimisation

✗ recompilation of non hot-spots
✗ large time overhead for large codes
✗ input-data needs to be processed at build time
  ⇒ typically a task of the compiled code
✗ no binary releases
Design Space

B. Pre-instantiate Code for all Cases

- generate code variants for sets of relevant parameters and **select at runtime**
  - e.g. template value-parameter specialisation
- fall-back default implementation
- performed by some compilers
  - e.g. vectorised (masked/unmasked, ...) and non-vectorised loop/function versions

✓ no runtime compilation complexity
✓ uses application code for input processing
× limited to small, discrete parameter domains
× limited to a small number of such parameters
× increased size of generated code
Design Space

C. Call a Compiler Library at Runtime

- compile hotspot code at runtime using a suitable library
  - OpenCL (intended for portability, own kernel language and runtime)
  - LLVM

✔ uses application code for input processing
✔ not limited by number of parameters/domains

× runtime overhead for compilation
× limited to the capabilities of the chosen library (i.e. LLVM)
  - LLVM lacks SIMD math functions
× porting to OpenCL is a major effort
Design Space

D. Call an Arbitrary Compiler at Runtime

- call a command line toolset
  - GCC, Clang/LLVM, Intel, Cray, ...
- create and load shared library

✓ uses application code for input processing
✓ not limited by number of parameters/domains
✓ use capabilities of any command line toolset

✗ larger runtime overhead for compilation

⇒ model of choice
Library: **KART - Kernel-compilation At Run Time**

- provide means for runtime compilation and invocation of arbitrary functions
- API for **C**, **C++**, and **Fortran** (implemented in modern C++)
- API similar to OpenCL, serves as a drop-in replacement for OpenMP applications
- use any compiler like on the command line
  - LLVM/JIT is not enough
  - need specific vendor optimisations (Intel, Cray, ...)
  - maximum flexibility

⇒ enables compiler optimisations based on runtime-data
  - conditionals, loops, memory access, vectorisation, ...
KART API concepts

- **program**
  - created from source code
  - can be built
  - contains kernels

- **toolset**
  - config files:

```plaintext
[compiler]
exe=/usr/bin/g++
options-always=-c -fPIC
options-default=-g -std=c++11 -Wall

[linker]
exe=/usr/bin/g++
options-always=-fPIC -shared
options-default=-g -Wall
```

- **export** KART_DEFAULT_TOOLSET=gcc.kart

- **kernel_ptr**
  - type-safe callable template
  - can be used like any function
toolset

- toolset(const string& config file name)
- get_compiler_options() : const string&
- set_compiler_options(const string&) : void
- append_compiler_options(const string&) : void
- get_linker_options() : const string&
- set_linker_options(const string&) : void
- append_linker_options(const string&) : void

program

- program(const string& src)
- create_from_file(const std::string& file_name) : program
- create_from_binary(const std::string& file_name) : program
- build(const toolset& ts) : void
- rebuild(const toolset& ts) : void
- get_build_log() : const string&
- get_kernel(sig T)(const std::string& name) : sig T
- get_kernel_ptr(sig T)(const std::string& name) : kernel_ptr(…)
- get_binary_file_name() : const std::string&

kernel_ptr

- operator(Args…) : Ret

1
0..*
0..1
0..*
KART Implementation

```
// kernel call site
kernel_foo = get_kernel(...);
kernel_foo(...);
... 4
5
```

The diagram illustrates the process of building a KART application:

1. The application input is processed.
2. Kernel source files are compiled.
3. The compiled kernel sources are linked with other components.
4. Dynamic symbols are loaded into the application.
5. The KART framework is initialized.

The application is linked against the dynamic library to access kernel functions.
KART C++ example

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

int main(int argc, char** argv) {
    /* ... application code ... */
    // call the kernel as usual
    double res = my_kernel(3.0, 5.0);
    /* ... application code ... */
}
# include "kart/kart.hpp"

// signature type
using my_kernel_t = double(*)(double, double);

// raw string literal with source
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 ... */
}
WIP: selecting runtime-compiled source via annotations

BEGIN_KART_COMPILED_CODE(my_kernel, double(*)(double, double))
double my_kernel(double a, double b)
{
    return a * b;
}
END_KART_COMPILED_CODE()

Idea:

- easier adaptation of existing code
- use preprocessor to generate wrapping code around functions
- kernel name and type are specified manually
  ⇒ can be enabled/disabled globally per define

Problem:

- edgy use of preprocessor
  - only works with "g++ -E", followed by compilation (not in a single command)
Benchmarks - Synthetic Kernels

```c
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;
}
```

```c
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;
    }
}
```
### Benchmarks - Synthetic Kernels

#### Synthetic kernel runtime comparison

<table>
<thead>
<tr>
<th>Operation</th>
<th>w/o KART</th>
<th>KART</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>matvec, alpha=0, HSW</td>
<td>4.67 x</td>
<td>2.58 x</td>
<td>7.86 x</td>
</tr>
<tr>
<td>matvec, alpha=1, HSW</td>
<td>2.61 x</td>
<td>2.58 x</td>
<td>4.67 x</td>
</tr>
<tr>
<td>convolve, off=0, HSW</td>
<td>2.61 x</td>
<td>2.58 x</td>
<td>4.67 x</td>
</tr>
</tbody>
</table>

#### Diagram

![Synthetic kernel runtime comparison graph](image-url)
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.
HEOM - Hierarchical Equations of Motion - is a model for computing open quantum systems, e.g. used to simulate energy transfers in photo-active molecular complexes.
Compilation Overhead

Runtime compilation techniques pay off when the accumulated runtime savings of all kernel calls exceed the runtime compilation cost.

- speed-up of the runtime-compiled kernel over the reference kernel:
  \[ s_b = \frac{t_{\text{ref}}}{t_{\text{kart}}}, \quad t_{\text{ref}} > t_{\text{kart}} \Rightarrow s_b > 1 \]

- \( s_b \) is an upper bound for the actual speed-up \( s \) including compilation overhead, where \( n \) is the number of kernel runs:
  \[ s = \frac{n \cdot t_{\text{ref}}}{n \cdot t_{\text{kart}} + t_{\text{compile}}} \]

- number of calls \( n_c \) needed to amortise \( t_{\text{compile}} \):
  \[ n_c = \frac{t_{\text{compile}}}{t_{\text{ref}} - t_{\text{kart}}} \]
Compilation Overhead

Runtime compilation techniques pay off when the accumulated runtime savings of all kernel calls exceed the runtime compilation cost.

- speed-up of the runtime-compiled kernel over the reference kernel:
  \[ s_b = \frac{t_{ref}}{t_{kart}}, \quad t_{ref} > t_{kart} \Rightarrow s_b > 1 \]

- \( s_b \) is an upper bound for the actual speed-up \( s \) including compilation overhead, where \( n \) is the number of kernel runs:
  \[ s = \frac{n \cdot t_{ref}}{n \cdot t_{kart} + t_{compile}} \]

- number of calls \( n_c \) needed to amortise \( t_{compile} \):
  \[ n_c = \frac{t_{compile}}{t_{ref} - t_{kart}} \]

HEOM:
\[ n_c \approx 10^3, \quad n_{90} \approx 10^4, \quad n \approx 10^5 \]
Benchmarks - Compile Time

**HEOM kernel compilation cost**
- **empty kernel**
- **auto vect. kernel**

- **OpenCL, HSW**: 56 ms / 95 ms
- **KART+intel, HSW**: 2337 ms / 2987 ms
- **KART+gcc, HSW**: 1990 ms / 2065 ms
- **KART+clang, HSW**: 1893 ms / 2130 ms
- **OpenCL, KNL**: 176 ms / 500 ms
- **KART+intel, KNL**: 7734 ms / 11110 ms
- **KART+gcc, KNL**: 7872 ms / 8152 ms
- **KART+clang, KNL**: 10272 ms / 10004 ms

**Compile time [ms]**
- 0 2000 6000 10000 14000
Goal: Reduce compile time overhead

Ideally:

- Standardised library API provided by compilers
  - ⇒ no processes
  - ⇒ no file operations
  - ⇒ no network operations (e.g. license server)
- OpenMP directives (with same compilers)
Goal: Reduce compile time overhead

Next steps:

- add **LLVM/MCJIT** as backend (approach C.) to save compile time where LLVM yields sufficient code
  ⇒ see how much overhead remains (without process creation and file I/O)
Goal: Reduce compile time overhead

Next steps:

- add **LLVM/MCJIT** as backend (approach C.) to save compile time where LLVM yields sufficient code
  
  ⇒ see how much overhead remains (without process creation and file I/O)

- implement **automatic kernel cache**
  
  ⇒ cache the generated libs with checksums based on source, toolchain, and options
  
  ⇒ similar to PoCL (OpenCL implementation using the LLVM toolchain like KART)
Goal: Reduce compile time overhead

Next steps:

- add **LLVM/MCJIT** as backend (approach C.) to save compile time where LLVM yields sufficient code
  ⇒ see how much overhead remains (without process creation and file I/O)

- implement **automatic kernel cache**
  ⇒ cache the generated libs with checksums based on source, toolchain, and options
  ⇒ similar to PoCL (OpenCL implementation using the LLVM toolchain like KART)

- compilation **server/deammon**
  ⇒ global kernel-cache (more re-use)
  ⇒ compile fast on Xeon, run fast on Xeon Phi
  ⇒ limit license use
Runtime compilation allows much more

- **benchmarking/auto-tuning** of kernels based on input data
- can be combined with source **code generation** techniques
- different variants of the same kernel
  - even from different compilers/versions
- single binary for different SIMD instruction sets (even unknown ones)
- cross language use
- ...
Runtime compilation allows much more

- **benchmarking/auto-tuning** of kernels based on input data
- can be combined with source **code generation** techniques
- different variants of the same kernel
  - even from different compilers/versions
- single binary for different SIMD instruction sets (even unknown ones)
- cross language use
- ...

Example

- benchmark math function on HLRN-III Cray XC40 supercomputer
  1. host application compiled with Cray compiler
  2. generates benchmark kernel source from template
  3. compiles and links in code with Cray, Intel, Clang, and GCC
  4. benchmarks kernels

⇒ ...and it works!
EoP - Thank you!

- The code will be available soon:
  - [https://github.com/noma/kart](https://github.com/noma/kart)
  - click "Watch" and wait
  - or send me a mail
  - Boost Software License (BSD/MIT-like)

- Questions, use cases, ideas, ...?
  - contact me: noack@zib.de

- Paper: