Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

KernelGen – the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs

Dmitry Mikushin[1], Nikolay Likhogrud[2], Eddy Zheng Zhang[3], Christopher Bergström[4]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] Lomonosov Moscow State University, Russian Federation
[3] Department of Computer Science, Rutgers University, USA
[4] PathScale Inc.

Abstract

GPUs are becoming pervasive in scientific computing. Originally served as peripheral accelerators, now they are gradually turning into central computing nodes. However, most current directive-based approaches for parallelizing sequential legacy code such as OpenACC and HMPP simply off-load "hot" CPU code onto GPUs, entailing a lot of limitations such as unsupported external calls and coarse-grained data dependence analysis. This paper introduces KernelGen, which is a parallelization framework with robust parallelism detection mechanism and a novel GPU-centric execution model. KernelGen supports the major scientific programming languages including C and Fortran, and has multiple backends that can generate target code for both X86 CPUs and NVIDIA GPUs. The efficiency of KernelGen has been demonstrated by the performance improvement up to 5.4x compared with three major commercial OpenACC compilers over a benchmark suite of numerical kernels.

Report Info

Published
July 2013
Number
USI-INF-TR-2013-2
Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
Online Access
www.inf.usi.ch/techreports

1 Introduction

To take advantage of massive deployment of GPU-enabled computing clusters, we need to conduct translation on a broad range of legacy scientific applications (mostly sequential) into GPU code. The CUDA and OpenCL programming models are well suited for re-writing small programs with a few intensively used computational kernels. However, for larger applications consisting of many individual interacting blocks, such as applications in the numerical models, the complexity of setting up efficient interaction between different building blocks manually to generate new GPU code increases dramatically. Many companies and research groups are postponing the porting process of their applications into GPU code. In addition to GPU-specific implementation, they often require to keep the conservative CPU version, which adds extra overhead in the development and support of segmented code base. Moreover, scientific specialists who are used to working with simple CPU code in Fortran in their problem domain, could hardly deal with complicated details of GPU parallelism, which further limits their research productivity. Driven by the need to simplify this process of programming GPUs for different problem domains, a number of new programming models/paradigms have been proposed and implemented:

- **Directive-based extensions to existing high-level languages with user-annotated parallelism.** This type of programming technologies introduce sets of directives (annotations) for marking up the code regions intended for execution on GPU. Based on this information, compiler automatically generates

hybrid executable binary. In order to standardize the set of directives for C/C++/Fortran, commercial compiler vendors formed OpenACC [1] and OpenHMPP [4] consortiums. The F2C-ACC source-to-source processor [7] is able to perform directive-based GPU code generation for a subset of Fortran programming language. Similar set of directives is being developed by Intel for Many Integrated Core (MIC) platform [3].

  Despite the expressiveness of high level code to be parallelized, directive-based extensions still require notable developer effort in organizing correct and efficient computations. Some of aforementioned compilers perform extra checks to ensure loops parallelism and transformation correctness, others – blindly follow the user preference. Compilers are often "over conservative" while making decisions about loops parallelism based on internal analysis, and user may need to force parallelization with additional directives (for example, "loop independent" directive of OpenACC). Most of directive-based compilers do not support generation of GPU kernels for loops with calls to functions from other object files or external libraries, leading to severe limitations of translating large structured applications into reusable GPU code modules.

- **Domain-specific languages (DSL) designed to express the parallelism of algorithms in specific problem domain.** In recent years, many DSLs and embedded DSLs have emerged. The main idea is to bring the language features and specific problem domain features closer, and meantime – avoiding strict specialization for any particular hardware. DSLs introduce an additional level of programming abstraction, which is translated by compiler or source-to-source processor into specific target architecture code. For instance, PATUS [6] is a C-like DSL designed for programming finite difference problems on rectangular grids. Efficient code could be generated for multicore CPUs with support of SSE and AVX extensions. Another DSL library called Stencil++ [9] is proposed for the similar problem domain, but embedded into C++ and templates are used extensively. An embedded DSL called Halide [13] supports code generation for x86-64/SSE, ARM v7/NEON and NVIDIA GPUs, and is intended mainly for image processing.

  The evaluation of DSLs/eDSLs is usually performed in comparison to hand-tuned programs, making it hard to evaluate the possible benefits over automatic directive-based compilers and other DSLs. Every unique DSL typically has only one development group, limiting its growth and evolution. Utilizing DSLs in existing programs usually require massive code rewriting, which falls back into the similar problems for CUDA and OpenCL discussed earlier.

- **Automatic analysis to extract parallelism based on polyhedral models.** This type of techniques are intended for detecting data dependencies within loop iteration spaces with exact methods or heuristics. Heuristics are currently utilized by most of commercial compilers, while research and experimental solutions often implement more complex methods, such as the *polyhedral analysis*. For instance, [11] implemented a GCC compiler extension for automatic transformation of parallel loops into OpenCL kernels, using CLooG polyhedral analysis library [5]. Another similar solution capable of transforming C loops into CUDA kernels is called PPCG [18]. Source-to-source compiler Par4all [17] transforms C and Fortran code into CUDA, OpenCL or OpenMP kernels using another polyhedral analysis system called PIPS. However, these models mainly focus on specific loops parallelization and do not have sophisticated enough techniques to handle communication between CPUs and GPUs, which has become performance bottleneck for most scientific applications.

In any case, explicit CUDA/OpenCL, directive-based programming models or DSLs require significant amount of manual code modification. For this reason, it is practically very difficult to completely port a large scale sequential application to GPUs. If an application is only partially ported, the host-device data synchronization may significantly impact the overall performance. For example, when porting only single WSM5 block of the WRF [15] model with the PGI Accelerator, the time spent on data synchronization is 40-60% of the total time [20].

Considering the limitations of existing technologies for porting large scientific applications into massively parallel architectures, we would like to have a number of desirable properties of next-generation parallelization compiler framework:

- Support a large set of *existing* popular programming languages;
- Automatically extract parallelism and transform the code into GPU code;

- Code generation process, integration of both GPU and host code;
- Minimize the data communication between the main system memory and the GPU;
- Allow the co-existence with other levels of parallelism, for instance, MPI.

In this paper, we propose KernelGen, a compiler and runtime prototype that targets at all these above requirements. We build KernelGen based on existing LLVM infrastructure and some research tools for automatic loops analysis. We would like to summarize the main contributions/novelties of this paper as follows:

- Automatic compilation of unmodified sequential C/Fortran program code into mixed CPU+GPU binary;
- Runtime data dependency analysis and JIT-compilation of GPU code;
- Language features: parallelization of some goto- or while-loops, loops with pointer arithmetics, implicit loops from Fortran array-wise statements and elemental functions;
- Execution model based on original dynamic code loader and linker, involving deep knowledge of Fermi/Kepler GPUs ISA;
- Open-source compiler pipeline (with exception of CUDA runtime and driver stack) assembled from GCC frontends, LLVM-based middle-end and NVPTX backend.

We compare KernelGen with three major commercial OpenACC compilers. We demonstrated the efficiency of KernelGen by up to 5.4x speedup over these commercial compilers and at least similar performance for the other benchmarks. The rest of the paper is organized as follows: Section 2 describes the compiler pipeline, linking, execution model and memory management; Section 3 explains the necessary modifications made to existing parallel loops analysis and transformation tool, in order to generate GPU kernels; Sections 4 and 5 are dedicated to auxiliary compiler subsystems and performance evaluation respectively.

2 KernelGen compiler pipeline

During compiler toolchain development, it is important to choose the most suitable existing infrastructure, by a number of criteria: existing frontends for different languages, flexibility of internal representation, presence of the basic optimization passes and efficient backends for target architectures, popularity and community support. The most suitable candidates are GCC, LLVM [12] and Open64 compilers. GCC compiler supports the highest number of programming languages, but does not have GPU frontends, while LLVM and Open64 have backends for NVIDIA PTX ISA. Open64 compiler has frontends for C, C++ and Fortran, generates fairly efficient code, but unfortunately has very segmented community. LLVM compiler does not have Fortran frontend, but DragonEgg [14] plugin can bridge GCC frontends with LLVM middle-end and backends. LLVM has its own NVPTX GPU backend, features simple intermediate representation (LLVM IR) and is developed much more intensively than GCC or Open64. Driven by these considerations, KernelGen is based on LLVM.

Although compatibility is extremely important to support large complex applications, it is often neglected in the novel programming models design. KernelGen compiler works directly with the original application, no changes in the source code or in the compilation process are required. Technically, KernelGen chains as a plugin to a slightly modified GCC compiler frontend, and therefore is fully compatible with its command line options. From the user point of view, this means a program configured to compile with *gcc* or *gfortran* could be simply switched to *kernelgen-gcc* or *kernelgen-gfortran*. A hybrid executable created by KernelGen will contain both CPU-only and GPU-enabled binaries. Depending on the value of *kernelgen_runmode* environment variable, either CPU or GPU version of application could be launched.

In order to conserve the original build process, a multi-stage pipeline similar to Link Time Optimization (LTO) is used: the preliminary representation of GPU code is first embedded into the special section of object files and then is transformed into GPU kernels source during linking. The final compilation of GPU kernels into binary code is performed by request in runtime (JIT, just-in-time compilation). The basic flowgraph of KernelGen compiler pipeline is shown in Fig. 1.

As the result, the original application is converted into the set of GPU kernels: one (for executable) or more (for multiple shared libraries) *main* kernels, and many *computational loops* kernels. The main kernels are executed on GPU in single thread. They are intended to track the static and dynamic data, execute

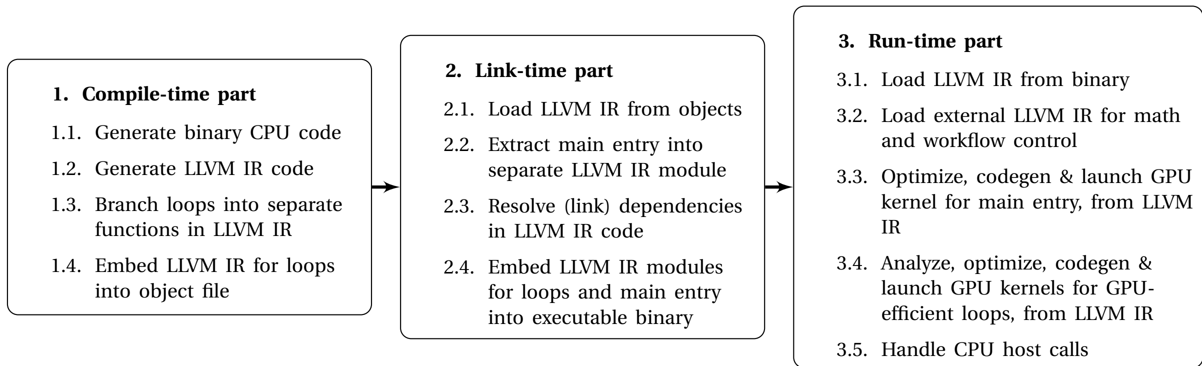


Figure 1: KernelGen compiler pipeline

some simple serial code, launch computational kernels on GPU and offload onto CPU the non-portable host functions calls, as well as the code portions inefficient for GPU execution. While the main kernels are serial, the computational loops kernels are executed in parallel to completely utilize the GPU resources. Thus, the largest possible portion of code is executed on GPU, while CPU only coordinates kernels interaction. For instance, in MPI application compiled with KernelGen each process will run a main GPU kernel, a set of computational kernels and some hostcalls for MPI routines. CUDA-aware implementation of MPI [16] may additionally eliminate CPU-GPU data roundtrips, if messages could be passed between GPUs in peer-to-peer mode.

KernelGen execution model has a lot of common with Intel MIC native mode, but works on GPUs, where scalar multiprocessors could be efficiently deployed, without the need of code vectorization.

2.1 Compilation

During individual objects compilation both x86-assembler and LLVM IR are generated, thus the application could still be deployed on host without GPUs. In order to parse the source code, the GCC compiler frontends are used together with the DragonEgg plugin, converting the GCC's *gimple* into LLVM IR. In LLVM IR of each object the computational loops are extracted into separate functions callable through the generic *kernelgen_launch* interface:

```
int kernelgen_launch(
    char* kernel,
    unsigned long long szdata,
    unsigned long long szdatai,
    unsigned int* data)
```

where *kernel* is the function name (in runtime is replaced with the fixed function address), *szdata* and *szdatai* are the sizes of function arguments and integer function arguments respectively (integer arguments are used as a signature for searching precompiled kernel binaries in runtime), and *data* are the function arguments aggregated into naturally aligned structure.

Stacks of nested loops are extracted into separate functions using the LLVM *BranchedLoopExtractor* pass in compile-time. As result of this pass, each loop in LLVM IR is cloned into its own function (that may later become a GPU kernel), execution switches between original version and function call, depending on the result of branching instruction:

```
if (kernelgen_launch(kernel, szdata, szdatai, data) == -1) {
    // Launch original loop.
}
```

With such conditional construct the KernelGen runtime library is able to switch between different loop representations. For instance, if the particular loop is identified as non-parallel, the *kernelgen_launch* returns -1, and the main kernel executes its original code. This loop may still have nested parallel loops, which will be recursively visited in the same fashion. If the whole stack of nested loops is non-parallel, or makes calls to unresolved external CPU functions, or is estimated to have no GPU execution benefit, then it is executed on host using the *kernelgen_hostcall* interface:

```
__device__ void kernelgen_hostcall(
    char* kernel,
    unsigned long long szdata,
    unsigned long long szdatai,
    unsigned int* data);
```

In case of the host call request, the main GPU kernel issues a callback to host, passes function name and argument list and suspends until the host call is finished. Host compiles and executes the requested function using the Foreign Function Interface (FFI).

Fortran code compilation has a useful side-effect of GCC-based frontend: in GIMPLE IR many high-level constructs are expanded into explicit loops. For instance, array-wise statements and elemental functions will end up as plain loops in LLVM IR:

```
complex*16, allocatable, dimension(:) :: c1, c2, z
...
z = conjg(c1) * c2
```

This way KernelGen can parallelize and port to GPU implicit loops, which are not covered by the current directive-based approaches.

2.2 Linking

When linking individual objects in the resulting application or library, LLVM IR code also gets linked into IR-module for main kernel, and one IR-module for each individual loop kernel – completely optimized and inlined. At the end on linking, IR code is embedded into application binary and then optimized and compiled in runtime, on demand.

Special care must be taken of the global variables and constants. Although global values are located in GPU global memory, they could not be shared across kernels compiled into separate object files, due to the absence of GPU dynamic linker. To workaround this issue, all global values uses are indexed during linking and replaced with actual addresses in runtime at LLVM IR level.

2.3 Execution model

The main kernel is launched with application startup and runs on GPU all the time. When host call or computational kernel is executed, the main kernel suspends (actively spins on atomic CAS) and continues execution only after the callback is finished. To work with this design, GPU must support the concurrent kernels execution (CKE) or kernel preemption. CKE is supported by NVIDIA GPUs with Compute Capability 2.0 and higher, while on AMD GPUs this feature is not supported. For this reason KernelGen currently works only with NVIDIA GPUs.

The *kernelgen_launch* and *kernelgen_hostcall* calls consist of two parts: GPU device-functions and CPU calls. These functions perform the final binary code generation and GPU kernel launch or arguments loading and CPU function launch using FFI, respectively. The message passing between host and device parts can be organized through GPU global memory or host pinned memory. To guarantee the correct values delivery, read and write operations must be *atomic*. For this reason, the interaction has been implemented using the global memory.

On Kepler K20 GPUs previously compiled kernels can be launched right from the main kernel, without CPU host call, using the feature of dynamic parallelism.

Since the main GPU kernel makes calls to other computational kernels, it must be able to pass any of its data as their arguments. But the data allocated in local memory cannot be shared between kernels. To workaround this limitation, the original NVPTX backend was modified to host local variables in *.global* PTX data section, making them shareable across all GPU kernels and host.

2.4 Memory management

One of the unique design solutions behind KernelGen is the memory management subsystem. Initially, the whole application data is kept in GPU memory, along with the code. In order to make CPU functions calls compatible with this concept, the memory synchronization layer is introduced. Once the CPU function tries to access the address in the GPU memory range, the segmentation fault signal handler maps the GPU

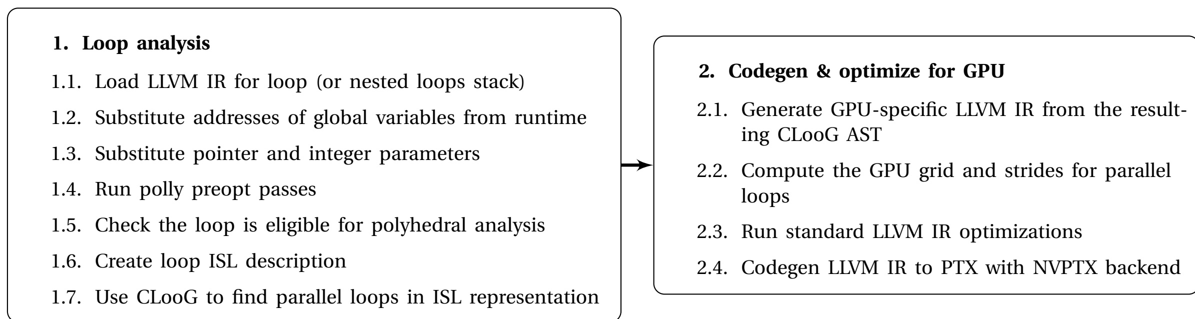


Figure 2: Loops analysis and parallel GPU code generation pipeline in KernelGen compiler. Optimization is performed for entire SCoP, code generation – for each individual function

memory pages into CPU tables and copies the input data. After the host call is finished, the "dirty" pages are synchronized back with the GPU.

Memory synchronization is limited to use only page-aligned mapping (4096 bytes), therefore having all data items aligned by page boundary would be a very convenient simplification at this moment. Unfortunately, current CUDA runtime (5.0) ignores the alignment settings. In order to workaround this issue, all data items are padded to page size at CUBIN level, using *libelf* library functions.

3 Generating GPU kernels for parallel loops

KernelGen performs the final step of LLVM IR analysis and binary GPU code generation in runtime, right before the corresponding code region is approached during program execution, similar to JIT-compilation. Such an approach is used to strengthen assumptions about data dependencies, based on the actual values of pointers and loops dimensions.

3.1 Runtime context substitution & analysis

Upon the first GPU kernel launch, only LLVM IR representation of code region exists. Each launch is performed with an aggregated structure of pointer and value arguments. First, integer values and pointers are substituted into LLVM IR. Additional LLVM pass transforms exact memory accesses into ISL form, compatible with Polly and CLooG. Once unknown parameters are eliminated, polyhedral transformations of loops no longer depend on poor compile-time alias analysis information and can reliably determine if certain memory ranges are intersecting. This method solves a usual issue of computational functions with parameter input and output arrays of unknown origin, which compiler has to mark "may alias" and consider even simple loop as potentially having dependent iterations. In OpenACC such cases can only be handled manually, either by specifying *restrict* attribute for pointers or "loop independent" directive for loops.

3.2 Polyhedral analysis

Polly [8] (from the polyhedral analysis), a part of LLVM infrastructure, is a set of loops transformation passes based on CLooG [5]. It is able to identify the parallel loops in LLVM IR, add extra small loops (tiles) for more efficient caching, perform loops interchanging, and map loops onto multiple CPU threads with OpenMP. For the given source code CLooG builds an *abstract syntax tree* (AST), and splits some fused loops. Thanks to splitting, the equivalent partially parallel representation could be carried out even for loops that were originally non-parallel. Such approach is rarely used, most of the modern compilers only check the existing loops parallelism without deep analysis.

Polly works with the parts of program, whose control flow and memory access patterns could be predicted, depending on the fixed set of parameters. Such parts are called *static control parts* (SCoPs) The part of a program is a SCoP if the following conditions are met:

1. SCoP contains only for-loops and if-conditions:

(a) Each loop has a single integer induction variable incremented from a lower bound to an upper bound by a unit stride. Upper and lower bounds are affine expressions of SCoP-independent integer parameters and induction variables of parent loops;

(b) Expressions in if-conditions must be affine and may depend on SCoP parameters and induction variables.

2. Memory accesses are performed using offsets applied to pointer-parameters of SCoP. Offsets are affine expressions of SCoP parameters and induction variables;

3. Only calls to functions without side effects are allowed within SCoP.

The first condition implies the structured control flow: it shall be possible to logically split the code into a hierarchy of single-entry, single-exit fully enclosed basic blocks. Instructions breaking the control flow (*break*, *goto*) are not allowed. Given affine expressions and SCoP parameters, Polly can use methods of linear programming to compute the loops boundaries and memory accesses patterns.

If Polly had worked with program in high-level language (AST) directly, many constructs such as *pointer* arithmetics, *while* loops or *goto* operators would have violated the above requirements. In LLVM IR, an assembler-level language Polly works with, *pointer* arithmetics is lowered into register operations, and any loop, regardless its type (*for*, *while*) is implemented uniformly, as conditional branch. As a consequence, Polly has two nice features:

- parallelize some *goto*- and *while*-loops, which is not supported by the OpenACC standard;
- parallellize loops with *pointer* arithmetics, which is unsupported at least in PGI OpenACC.

During adaptation of Polly for GPU kernels generation, the existing OpenMP code generator has been partially reused. In OpenMP case, if the outer loop is parallel, then its body is wrapped into separate function and is called through the *libgomp* – GNU OpenMP implementation. The mapping of loop iterations on CPU threads is performed by OpenMP runtime, and only the most outer loop is parallelized. For KernelGen this logic was modified in the following ways:

1. Not only the outer loop, but all nested loops are processed recursively, to utilize the multidimensional GPU compute grids;

2. Iteration space of each loop (up to 3D) is mapped onto GPU compute grid, favoring coalesced GPU global memory transactions.

Suppose in a given nested loops stack it is possible to parallelize N closely-nested loops. Then the kernel can be launched on a grid with N dimensions (for CUDA $N \leq 3$). For each dimension mapped onto GPU threads KernelGen generates code to compute the thread index in block and block index in grid. Each parallel loop corresponds to single grid dimension in reverse order: the most inner loop corresponds to X dimension (this allows to coalesce memory transactions of threads in the same warp). For each parallel loop KernelGen generates code to determine the lower and upper boundaries of the iteration space executed by GPU thread. Finally, the code for loops with modified boundaries and strides is generated.

Fig. 2 shows loops analysis and parallel GPU code generation pipeline of KernelGen compiler.

4 Extra runtime subsystems

4.1 GPU math module

Introduction of LLVM backend for generating GPU assembly (NVPTX backend) was positioned by NVIDIA as “open-sourcing” the CUDA compiler. However, NVIDIA’s frontend for C/C++/CUDA is proprietary and closed-source, while clang supports only a very limited subset of CUDA keywords. Another significant part of compiler unavailable in LLVM is the GPU C99 math functions library. Since in CUDA compiler these functions are implemented as C/C++ headers, their use with other languages available in LLVM is problematic, with exception of a subset of builtins. For instance, such functions as double-precision *sin*, *cos*, *pow* are not available. In KernelGen this problem has been solved by converting standard C/C++ CUDA math headers into LLVM IR either using clang (required numerous code modifications, resulting IR code is possibly not

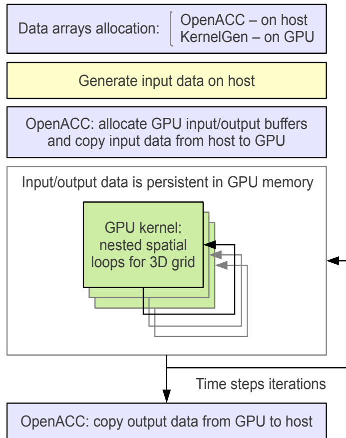


Figure 3: Organization of numerical programs in KernelGen performance test suite.

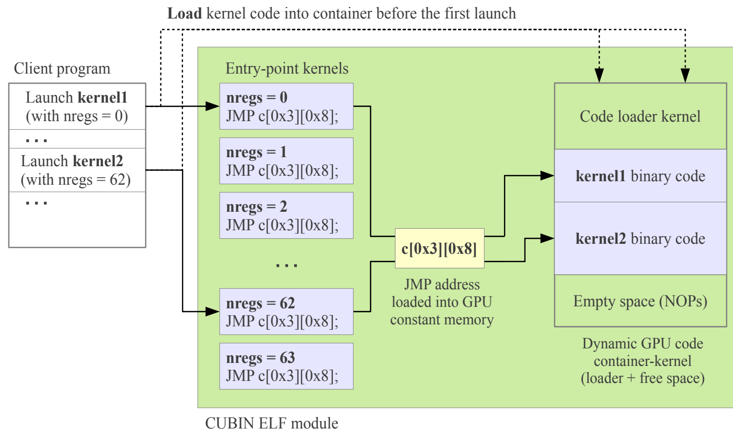


Figure 4: KernelGen custom dynamic kernels loader: new kernels code is loaded into dummy container-kernel address space and executed through one of 63 kernels-stubs denoting the register count and kernel code starting address.

entirely valid) or by dumping IR from cicc (a part of nvcc CUDA compiler pipeline). In latter case, IR math module could be produced by compiling an empty .cu-file and dumping IR code from cicc using debugger. IR code generated by cicc is compatible with the actual LLVM version and allows to perfom IR-level linking of client application code and math functions module, regardless the used original high-level language. For instance, using this method KernelGen is able to generate GPU kernels for Fortran programs.

4.2 Asynchronous GPU kernels loader

KernelGen needs to compile kernels in runtime and load their binaries into the GPU memory in the background of launched main kernel. Normally, manual kernels loading could be performed with *cuModuleLoad* and *cuModuleGetFunction* functions of standard CUDA Driver API. But both of these calls are implicitly synchronous, probably due to the memory allocation. Facing this problem, KernelGen had no way, but to provide its own implementation for loading kernels code into preallocated GPU memory region.

The kernel loader is based on the following concept. Initially, a large empty kernel (containing NOPs) is loaded into GPU memory with regular CUDA Driver API functions, to act as a container for other kernels code. Once runtime needs to load a new kernel, its binary code is copied into container address space, which is known through the Effective Program Counter value (LEPC instruction of Fermi ISA). This way container can host the code of many smaller kernels one after another (with some extra offset for proper instruction cache flushing). In fact, such dynamic kernels loader generally works as a simple memory pool. But there is also one extra characteristic to track: the register count. Dynamic loader creates 63 kernel stubs (entry points) using 1 to 63 registers. Once particular kernel launch is requested for the first time, its code is loaded into container, using device-side memory copying kernel (kernel loader), since *cudaMemcpy* will not permit copying to unmanaged memory range. Then, the entry-point kernel with matching register count, the specified GPU compute grid and code address is launched. The only instruction of entry point kernel performs jump to the start of actual kernel code specified in the fixed item of GPU constant memory (Fig. 4). As result, new kernels launches are performed without calls to *cuModuleLoad* and *cuModuleGetFunction*. Since there are no LEPC, absolute JMP and NOP instructions in CUDA C or PTX assembler, kernel loader/container and entry points are implemented in Fermi ISA, using AsFermi assembler [10]. Entry points register counts are defined in the sections of CUBIN ELF binary and are also set by AsFermi.

The described method could be further generalized into a tool for dynamic GPU binary code modification.

4.3 GPU kernels dynamic linker

Loops kernels are expected to have no static GPU memory allocations (all data is passed over the aggregated parameters structure), but still may make calls to other device functions, for instance, GPU math. Thus,

kernel loader should either support loading of called functions or let all kernels to reuse functions coming with the main kernel module, where they always present to serve the fallback branches. The current *ptxas* assembler (converts PTX to GPU ISA, e.g. to Fermi ISA) supports two modes for device functions calls:

- *cloning=yes* – in this mode every device function will have multiple copies, each one specialized for particular call site, which usually allows to perform more efficient register allocation for the price of larger code size. Cloned functions bodies follow the caller code and are accounted into the caller code size in CUBIN ELF records. Functions addresses used in calls are hard-coded in callers assembly. This mode is activated by default;

- *cloning=no* – in this mode a single copy of device function is shared across all call sites, possibly requiring more registers, but keeping the code very compact in comparison to cloned version. Functions addresses used in calls are unresolved (*JCAL 0x0*), ELF relocation table contains called functions names for the corresponding *JCAL* instructions offsets. Shared device functions are normal functions directly visible in CUBIN ELF symbols table or *cuobjdump*.

Some of GPU math functions have static memory allocations (e.g. trigonometric tables). For this reason, KernelGen shares functions across all kernels, using a mechanism, similar to conventional dynamic linking:

- all kernels are compiled with *cloning=no*;

- for loops kernels – only kernel code is loaded, for main kernel – kernel body, functions and the corresponding data;

- kernel loader and main kernel are merged into single module, to have all kernels in the same module

- upon main kernel load, KernelGen custom dynamic linker builds a table of functions names and their absolute addresses;

- upon loop kernel load, KernelGen custom dynamic linker resolves functions calls defined by the relocation table with (name, address) table of main kernel, replacing *JCAL 0x0* instruction with a *JCAL* to actual function address, using AsFermi assembler.

4.4 GPU dynamic memory heap

Several types of CUDA API functions, such as device memory allocation or loading of GPU binary module always force synchronization of all asynchronous operations. Since KernelGen execution model requires persistent run of main GPU kernel during entire application lifetime, any additional memory allocation or CUDA module load would result into a deadlock. This behavior of standard CUDA functions forced KernelGen to introduce alternative implementations.

The host version of GPU memory allocation function is reasonably synchronous. But even the device malloc function appears to be synchronous, which is unexpected, since the memory buffers for individual threads should be preallocated. For this reason, KernelGen currently implements its own simple dynamic GPU memory pool.

5 Evaluation

KernelGen is being tested on three types of applications: behavior correctness tests, performance tests and user applications. Behavior tests are intended to track code generation issues, performance tests allow to spot performance regressions in comparison to earlier KernelGen builds and other compilers. In performance testing, preference is given to intercomparisons between KernelGen and other parallelizing compilers, rather than with hand-written GPU kernels, because it allows to better analyse compiler capabilities within its class of software.

Performance test suite consists of several typical single and double precision numerical algorithms on 2D or 3D regular grids. Each algorithm is performed in 2-3 parallel spatial loops enclosed into non-parallel time iterations loop (Fig. 3). KernelGen automatically recognizes parallel spatial loops inside non-parallel time iterations loop, while OpenACC compilers do this only with appropriate manually inserted OpenACC directives. Tests are partially adopted from [6], short descriptions are presented in Table 1. The test suite is

Table 1: KernelGen performance benchmark

Test	Description	dims	language	Test	Description	dims	language
divergence	divergence operator	3D	C	matmul	matrix-matrix multiplication	2D	Fortran
gameoflife	Conway's game of life	2D	C	matvec	matrix-vector multiplication	2D	C
gaussblur	Gaussian blur 25-point approximation	2D	C	sincos	$z := \sin(x) + \cos(y)$	3D	Fortran
gradient	gradient operator	3D	C	tricubic	tricubic interpolation	3D	C
jacobi	Jacobi method iterations	2D	Fortran	uxx1	approximation of second derivative	3D	C
lapgsrb	Laplace operator 25-point approximation	3D	C	vecadd	arrays sum	3D	C
laplacian	Laplace operator 7-point approximation	3D	C	wave13pt	13-point 2 levels in time explicit scheme for wave equation	3D	C

specially designed to perform comparisons with directive-based language extensions. The current version supports OpenACC and OpenMP extensions for MIC (KernelGen compiles the same code, ignoring all directives). Fig. 6 shows normalized performance differences between tests kernels compiled with KernelGen, PGI, CAPS and PathScale (PathScale currently supports only Fermi GPUs). The corresponding absolute execution times and register counts are listed in Table 2. Fig. 5 shows an extra intercomparison between KernelGen and CPU versions of the corresponding kernels.

GPU kernels performance is sensitive to compute grid block size. During KernelGen performance evaluation the block size of $\{128, 1, 1\}$ (or $\{128, 1\}$ for 2d tests) has been identified as fastest one for all tests. Similarly, OpenACC compilers should be able to set some good compute grid config, in case user has not specified it explicitly. PGI also uses $\{128, 1\}$ blocks by default, but both for 2d and 3d loops. Since the default CAPS configuration for compute grid is very inefficient, a manual gang/vector setup has been introduced in all tests.

GPU kernels produced by all evaluated compilers are generic with respect to loops dimensions, which means the same kernel code should be able to handle an arbitrary problem size. For this reason, generic kernels include additional checks and strides for processing multiple grid points in each thread, in case problem dimension is larger than the corresponding compute grid dimension. Thanks to runtime constants substitution and JIT-compilation, KernelGen is able to generate more efficient kernel code for particular problem domains, eliminating unnecessary loops and branches. Specialization is automatically turned off, if kernel is recompiled too often, falling back to generic version.

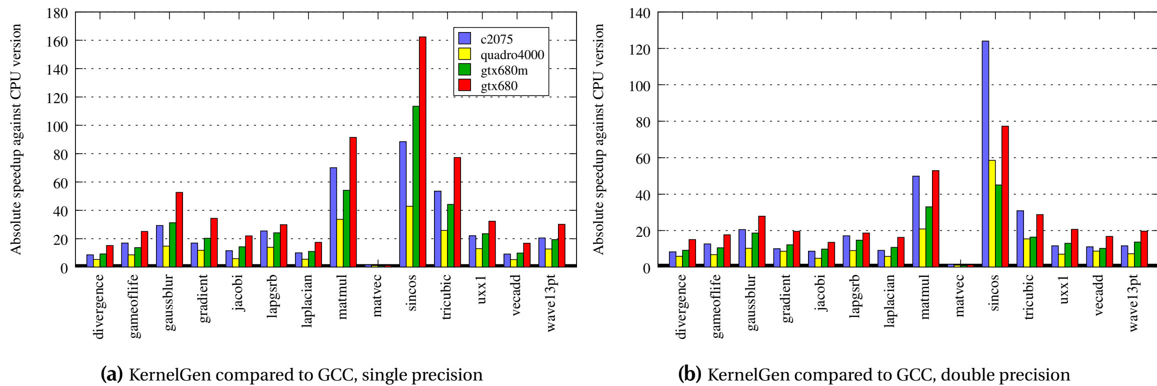
Tests *matmul* and *matvec* intentionally perform uncoalesced dot product as inner loops. In this case, compiler's best bet is to perform loop unrolling and fuse multiple loads into single wider load (e.g. four LD.32 into single LD.128), to consume as much memory bandwidth as possible. KernelGen only unrolls loops by factor of 3 and optimizes reduction, loads fusing is performed by ptxas. This optimization results into 1.5x-5.5x speedup over PGI and CAPS on K10 GPUs (Fig. 6e-6h).

Test *tricubic* uses a very large compute kernel to stress compiler capability to perform efficient register allocation and spilling. In some cases significant performance differences could be observed, for instance PathScale compiler is always slower (Fig. 6a-6d).

KernelGen uses LLVM IR module for GPU math functions, which is likely insufficiently optimized at this point. For this reason *sincos* test performance is often worse than OpenACC compilers. Official LLVM IR math module to be included into recently announced CUDA 5.5 release might be optimized better. Double-precision *sincos* does not work for some OpenACC compilers.

Overall, KernelGen is on par with all commercial compilers, and almost always shows the best performance on K10 GPUs (sm_30). While Tesla C2075 and Quadro 4000 performance footprints are very different (Fig. 6a-6d), GTX 680 and GTX 680M demonstrate very similar behavior (Fig. 6e-6h).

Additional testing of COSMO [2] and WRF [15] numerical models showed KernelGen is able to generate consistent GPU-enabled executables for complex applications in reasonable time.



(a) KernelGen compared to GCC, single precision

(b) KernelGen compared to GCC, double precision

Figure 5: Absolute speedup of test GPU kernels generated by KernelGen r1780 against CPU versions by GCC 4.6.3 on Intel Core i7-3610QM (for GCC). Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

6 Conclusion

KernelGen project implemented an original approach for automatic code porting on NVIDIA GPUs, well-suited for numerical applications. Conserving the original source code, compiler aims to move onto GPU the maximum possible portion of code, including memory allocations, creating efficient data layout principally for GPU computations. KernelGen performs loops parallelism analysis, based on Polly and CLooG, complementing them with GPU-specific LLVM IR code generation. LLVM IR is further lowered into PTX assembler using NVPTX backend, jointly developed by NVIDIA and LLVM community. Performance testing showed GPU kernels generated with KernelGen are on par with with three commercial OpenACC compilers.

In order to start broader use of KernelGen in scientific applications, some additional runtime subsystems still have to be implemented. For instance, the current version misses infrastructure for estimating kernels computational complexity and collecting execution statistics, needed for efficient dynamic switching between CPU and GPU versions. Parallel kernels generator should be extended to support tiling/locality for more efficient memory utilization, and reduction idiom recognition. Launching of loops kernels could be implemented more efficiently on K20 GPUs, where dynamic parallelism allows direct spawning another kernel launch from the current kernel, without a host call.

KernelGen source code is available under the University of Illinois/NCSA license (with exception of GCC plugin, which has to be GPL) at the project website: `http://kernelgen.org/`.

Acknowledgements

This work is supported by the Swiss Platform for High-Performance and High-Productivity Computing[1] (HP2C), testing is performed on cluster “Tödi” of Swiss National Supercomputing Centre (CSCS), cluster “Lomonosov” of Moscow State University [19] and on hardware donated by HP and NVIDIA.

References

[1] The OpenACC™ application programming interface. version 1.0. `http://www.openacc-standard.org`, Nov. 2011.

[2] Consortium for small-scale modeling. `http://www.cosmo-model.org/`, Dec. 2012.

[3] The heterogeneous offload model for Intel® many integrated core architecture. `http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf`, Dec. 2012.

[4] OpenHMPP, new HPC open standard for many-core. `http://www.caps-entreprise.com/openhmpp-directives/`, Apr. 2013.

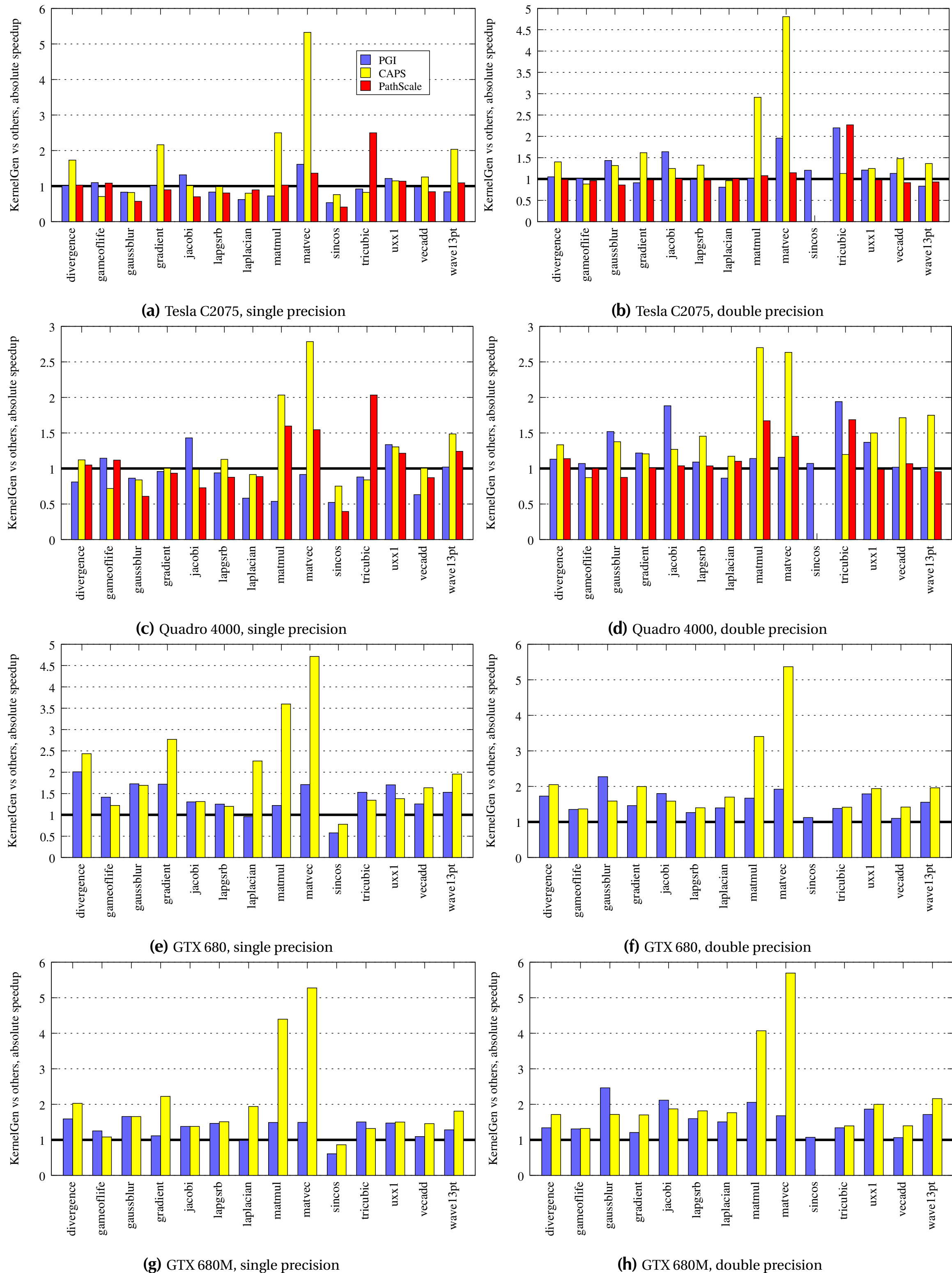
[5] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE PACT*, pages 7–16, 2004.

[1] `http://hp2c.ch`

Table 2: Absolute times (best time for test is marked blue) and register count of single precision test GPU kernels generated by KernelGen r1780, PGI 13.02, CAPS 3.2.4 and PathScale ENZO 2013 Beta on NVIDIA Tesla C2075 (Fermi sm_20), Quadro 4000 (Fermi sm_21), GTX 680 (Kepler sm_30) and GTX 680M (Kepler sm_30). Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

Test	**Tesla C2075**							**GTX 680**					
	KernelGen		PGI		CAPS		PathScale	KernelGen		PGI		CAPS	
	time	nregs	time	nregs	time	nregs	time	time	nregs	time	nregs	time	nregs
divergence	0.008463	18	0.008579	40	0.014654	22	0.008759	0.004810	20	0.009662	49	0.011712	32
gameoflife	0.010180	21	0.011178	28	0.007198	26	0.011088	0.006867	34	0.009714	32	0.008390	25
gaussblur	0.016010	56	0.013287	31	0.013146	26	0.009171	0.008880	51	0.015336	36	0.01503	34
gradient	0.010582	21	0.010745	42	0.022912	23	0.009555	0.005233	22	0.009000	51	0.014492	29
jacobi	0.007825	24	0.010314	23	0.007936	23	0.005524	0.004105	23	0.005348	30	0.005381	26
lapgsrb	0.017307	55	0.014387	61	0.017005	34	0.014023	0.014687	40	0.018369	63	0.017612	44
laplacian	0.008150	18	0.005071	39	0.006507	25	0.007318	0.004670	21	0.004461	48	0.010564	32
matmul	0.000994	16	0.000718	23	0.002466	22	0.001039	0.000548	16	0.000668	33	0.001947	28
matvec	0.016350	16	0.026357	22	0.085751	17	0.022241	0.020611	16	0.035250	25	0.097733	21
sincos	0.010796	22	0.005758	26	0.008202	22	0.004441	0.005845	34	0.003371	29	0.004549	26
tricubic	0.053463	63	0.049090	63	0.043981	47	0.133747	0.036861	61	0.056364	63	0.049621	54
uxx1	0.016480	32	0.020002	59	0.018898	41	0.018825	0.011258	32	0.019175	63	0.015535	44
vecadd	0.004838	12	0.004724	24	0.006068	17	0.004102	0.002623	12	0.003293	31	0.004293	18
wave13pt	0.011779	34	0.009886	54	0.023964	30	0.012920	0.007985	34	0.012204	60	0.015622	35

Test	**Quadro 4000**							**GTX 680M**					
	KernelGen		PGI		CAPS		PathScale	KernelGen		PGI		CAPS	
	time	nregs	time	nregs	time	nregs	time	time	nregs	time	nregs	time	nregs
divergence	0.013810	18	0.011191	40	0.015466	22	0.014484	0.007924	20	0.012595	49	0.016064	32
gameoflife	0.020142	20	0.023060	28	0.014457	26	0.022514	0.012555	21	0.015723	32	0.013589	25
gaussblur	0.031713	56	0.027388	31	0.026562	26	0.019262	0.014991	51	0.024847	36	0.024815	34
gradient	0.015171	21	0.014536	42	0.015217	23	0.014158	0.008855	22	0.009870	51	0.019691	29
jacobi	0.015162	24	0.021676	23	0.014963	23	0.011047	0.006325	23	0.008712	30	0.008722	26
lapgsrb	0.031717	55	0.029754	61	0.035739	37	0.027774	0.018310	40	0.026814	63	0.027654	44
laplacian	0.014693	18	0.008560	39	0.013432	25	0.013007	0.007431	21	0.007325	48	0.014407	32
matmul	0.002780	14	0.001494	23	0.005653	22	0.002192	0.000731	16	0.001088	33	0.003221	28
matvec	0.042747	15	0.039113	22	0.118878	17	0.036021	0.028515	16	0.042524	25	0.150424	21
sincos	0.022276	36	0.011623	26	0.016747	22	0.008807	0.008421	22	0.005130	29	0.007263	26
tricubic	0.110995	63	0.097652	63	0.093125	47	0.225702	0.064770	61	0.097441	63	0.085468	54
uxx1	0.028040	33	0.037450	59	0.036532	41	0.033995	0.015521	32	0.022862	63	0.023283	44
vecadd	0.008435	12	0.005331	24	0.008467	17	0.007341	0.004455	12	0.004869	31	0.006491	18
wave13pt	0.019013	34	0.01939	54	0.028230	30	0.023578	0.012462	34	0.015987	60	0.022544	35



(a) Tesla C2075, single precision

(b) Tesla C2075, double precision

(c) Quadro 4000, single precision

(d) Quadro 4000, double precision

(e) GTX 680, single precision

(f) GTX 680, double precision

(g) GTX 680M, single precision

(h) GTX 680M, double precision

Figure 6: Absolute speedup of test GPU kernels generated by KernelGen r1780 against PGI 13.02, CAPS 3.2.4 and PathScale ENZO 2013 Beta on NVIDIA Tesla C2075 (Fermi sm_20), Quadro 4000 (Fermi sm_21), GTX 680 (Kepler sm_30) and GTX 680M (Kepler sm_30). Values above 1 – KernelGen version is faster than competitor's, values below 1 – competitor's version is faster than KernelGen. Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

[6] M. Christen, O. Schenk, and Y. Cui. Patus for convenient high-performance stencils: evaluation in earthquake simulations. In *SC*, page 11, 2012.

[7] M. Govett. Development and use of a Fortran → CUDA translator to run a NOAA Global Weather Model on a GPU cluster. http://gladiator.ncsa.uiuc.edu/PDFs/accelerators/day2/session3/govett.pdf, Apr. 2009.

[8] T. Grosser, H. Zheng, R. A, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly - polyhedral optimization in LLVM. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, Apr. 2011.

[9] T. Gysi. Stencil++ for HP2C Dycore. http://mail.cosmo-model.org/pipermail/pompa/attachments/20120306/079fadc1/DWD_HP2C_Dycore_120305.pdf, Mar. 2012.

[10] Y. Hou. AsFermi: An assembler for the NVIDIA fermi instruction set. http://code.google.com/p/asfermi/, Dec. 2012.

[11] A. Kravets, A. Monakov, and A. Belevantsev. GRAPHITE-OpenCL: Automatic parallelization of some loops in polyhedra representation. http://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=get&target=belevantsev.pdf, Oct. 2010.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[13] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32, 2012.

[14] D. Sands. Reimplementing llvm-gcc as a gcc plugin. http://llvm.org/devmtg/2009-10/Sands_LLVMGCCPlugin.pdf, Oct. 2009.

[15] W. C. Skamarock, J. B. Klemp, J. Dudhia, D. O. Gill, D. M. Barker, M. G. Duda, X. Y. Huang, W. Wang, and J. G. Powers. A description of the Advanced Research WRF version 3. Technical report, 2008.

[16] J. Squyres, G. Bosilca, S. Sumimoto, and R. vandeVaart. Open MPI state of the union. http://www.open-mpi.org/papers/sc-2011/Open-MPI-SC11-BOF-1up.pdf, Nov. 2011.

[17] M. Torquati, M. Vanneschi, M. amini, S. Guelton, R. Keryell, V. Lanore, F.-X. Pasquier, M. Barreteau, R. Barrère, C.-T. Petrisor, É. Lenormand, C. Cantini, and F. D. Stefani. An innovative compilation tool-chain for embedded multi-core architectures. In *Embedded World Conference*, Nuremberg, Germany, Feb. 2012.

[18] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[19] V. Voevodin, S. Zhumatiy, S. Sobolev, A. Antonov, P. Bryzgalov, D. Nikitenko, K. Stefanov, and V. Voevodin. Practice of "Lomonosov" supercomputer. *Open Systems*, 7, 2012.

[20] M. Wolfe and C. Toepfer. The PGI accelerator programming model on NVIDIA gpus part 3: Porting WRF. http://www.pgroup.com/lit/articles/insider/v1n3a1.htm, Dec. 2012.