2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

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

Dmitry Mikushin*, Nikolay Likhogrud†, Eddy Z. Zhang‡ and Christopher Bergström§
*Institute of Computational Science, University of Lugano, Switzerland, dmitry@kernelgen.org
†Lomonosov Moscow State University, Russian Federation, n.lihogrud@gmail.com
‡Department of Computer Science, Rutgers University, NJ, USA, eddy.zhengzhang@cs.rutgers.edu
§PathScale Inc., cbergstrom@pathscale.com

***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 a 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 5.4× compared with three major commercial OpenACC compilers over a benchmark suite of numerical kernels.**

***Keywords*-GPU; LLVM; OpenACC; JIT-compilation; stencils**

I. 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 rewriting 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. In addition to GPU-specific implementation, it is often required to maintain the conservative CPU version, which adds extra overhead in the development and support of a 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 technology introduce sets of directives (annotations) for marking up the code regions intended for execution on a GPU. Based on this information, the 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 [2] consortiums. Accelerators support has been also included into OpenMP 4.0. The F2C-ACC source-to-source processor [3] is able to perform directive-based GPU code generation for a subset of Fortran programming language. A similar set of directives is being developed by Intel for the Many Integrated Core (MIC) platform [4].
  Despite the expressiveness, directive-based extensions still require notable developer effort in organizing correct and efficient computations. Compilers are often "over conservative" while making decisions about loops parallelism based on internal analysis, and the user may need to force parallelization with additional directives (for example, "loop independent" directive of OpenACC). Most directive-based compilers do not support generation of GPU kernels for loops with calls to functions from other object files or external libraries, limiting GPU use in large structured applications.
- **Domain-specific languages (DSLs) 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 the compiler or source-to-source processor into specific target architecture code. For instance, PATUS [5] 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. An embedded DSL called

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.115

Halide [6] 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. Utilizing DSLs in existing programs usually requires 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.** These types 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, [7] implemented a GCC compiler extension for automatic transformation of parallel loops into OpenCL kernels, using the CLooG polyhedral analysis library [8]. Another similar solution capable of transforming C loops into CUDA kernels is called PPCG [9]. Source-to-source compiler Par4all [10] 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 a performance bottleneck for most scientific applications.

In any case, explicit CUDA/OpenCL, directive-based programming models or DSLs require a 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 a single WSM5 block of the WRF model with the PGI Accelerator, the time spent on data synchronization is 40-60% of the total time [11].

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;
- have a code generation process, integration of both GPU and host code;
- minimize the data communication between the main system memory and the GPU;
- allow the coexistence with other levels of parallelism, for instance, MPI.

In this paper, we propose KernelGen, a compiler and runtime prototype that targets all these above requirements. We built 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 the 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 5.4× speedup over these commercial compilers and at least similar performance for the other benchmarks. The rest of the paper is organized as follows: Section II describes the compiler pipeline, linking, execution model, and memory management; Section III explains the necessary modifications made to existing parallel loops analysis and transformation tools, in order to generate GPU kernels; Sections IV and V are dedicated to auxiliary compiler subsystems and performance evaluation, respectively.

II. KernelGen compiler pipeline

During compiler toolchain development, it is important to choose the most suitable existing infrastructure, using 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, and generates fairly efficient code, but unfortunately has a very segmented community. LLVM compiler does not have Fortran frontend, but DragonEgg [13] 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 the *kernelgen_runmode* environment variable, either the CPU or GPU version of the 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 the KernelGen compiler pipeline is shown in Fig. 1.

As a result, the original application is converted into a 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 the GPU in single thread. They are intended to track the static and dynamic data, execute some simple serial code, launch computational kernels on the GPU, and offload onto the CPU the nonportable 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 the GPU, while the CPU only coordinates kernels interaction. For instance, in an 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 [14] may additionally eliminate CPU-GPU data roundtrips, if messages could be passed between GPUs in peer-to-peer mode.

A. Compilation

During individual object compilation both x86-assembly and LLVM IR are generated; thus the application could still be deployed on the host without GPUs. In order to parse the source code, the GCC compiler front-ends 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 unified interface: the function name (in runtime it is replaced with the fixed function address), the total sizes of all and integer-only loop parameters (integer parameters are used as a signature for searching precompiled kernel binaries in runtime), and the pointer to the structure, which aggregates all loop parameters.

Groups of nested loops are extracted into separate functions using our *BranchedLoopExtractor* LLVM 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), and application may switch between the original version, and function call, depending on the result of the branching instruction. With such a construct the KernelGen runtime library implements conditional execution of different loop representations. For instance, if the particular loop is identified as nonparallel, 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 group of nested loops is nonparallel, or makes calls to unresolved external CPU functions, or is estimated to have no GPU execution benefit, then it is executed on the host. In the case of the host call request, the main GPU kernel issues a callback to the host, passes a function name and argument list, and suspends until the host call is finished. The 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, arraywise 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.

B. Linking

When linking individual objects in the resulting application or library, LLVM IR code also gets linked into the IR-module for the 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 the 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 a GPU dynamic linker. To work around this issue, all global value uses are indexed during linking and replaced with actual addresses in runtime at the LLVM IR level.

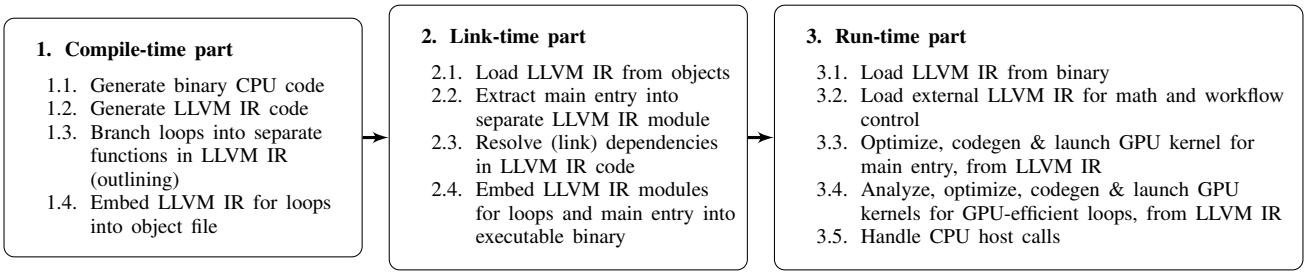


Figure 1: KernelGen compiler pipeline.

C. Execution model

The main kernel is launched with application startup and runs on the GPU all the time. When host call or computational kernel is executed, the main kernel suspends (actively spins on atomic CAS) and continues only after the callback is finished. To work with this design, the concurrent kernels execution is required, which is supported by NVIDIA GPUs since Compute Capability 2.0.

The launchers of extracted loop functions consist of two parts: GPU device-functions and CPU calls. These parts perform the final binary code generation and GPU kernel launch or arguments loading and CPU function launch using FFI, respectively. The synchronization between host and device parts is organized using a lock in GPU global memory.

D. Memory management

One of the unique design solutions behind KernelGen is the memory management subsystem. Initially, all the application data are 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 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.

III. 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.

A. Runtime context substitution and analysis

Upon the first GPU kernel launch, only an LLVM IR representation of the 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. An 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 typical issue of computational functions with parameter input and output arrays of unknown origin, which a compiler has to mark "may alias" and consider even a 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.

B. Polyhedral analysis

Polly [15] (from the polyhedral analysis), a part of the LLVM infrastructure, is a set of loops transformation passes based on CLooG [8]. 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 nonparallel. Such an 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.

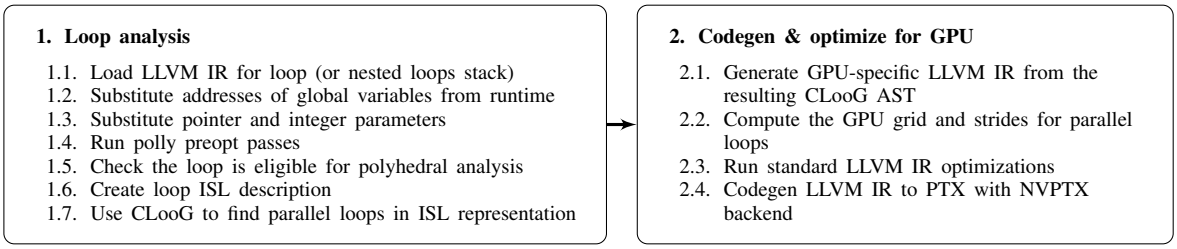


Figure 2: Loops analysis and parallel GPU code generation pipeline in KernelGen compiler. Optimization is performed for entire SCoP, code generation – for each individual function.

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* arithmetic is lowered into register operations, and any loop, regardless its type (*for*, *while*) is implemented uniformly, as a 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 not supported 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 a 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 outermost 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 three dimensions) is mapped onto a 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 a single grid dimension in reverse order: the innermost loop corresponds to X dimension (this allows one 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 the GPU thread. Finally, the code for loops with modified boundaries and strides is generated.

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

IV. Extra runtime subsystems

A. GPU math module

With the introduction of the LLVM backend for generating the GPU assembly (NVPTX backend), the NVIDIA CUDA compiler became partially open-source. NVIDIA's frontend for C/C++/CUDA remains proprietary, 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 the CUDA compiler these functions are implemented as C/C++ headers, their use with other languages available in LLVM is problematic, with the exception of a subset of built-ins. For instance, functions such as double-precision *sin*, *cos*, *pow* are not available. For KernelGen the math headers have been converted into LLVM IR module by tracing *cicc* (a part of the *nvcc* CUDA compiler pipeline). As demonstrated by performance test suite (see Section V), the resulting module allows one to perform IR-level linking of client application code with the math functions, regardless of the high-level language used. The proposed approach has been recently used by NVIDIA to generate official LLVM math modules (libdevice*), which became a part of CUDA 5.5 release.

B. Asynchronous GPU kernels loader

KernelGen needs to compile kernels in runtime and load their binaries into the GPU memory in the background of the launched main kernel. Normally, manual kernel 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 CUDA Driver API functions, to

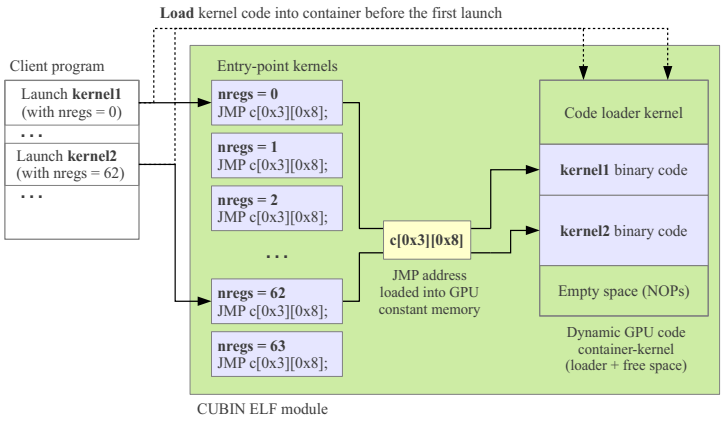


Figure 3: KernelGen custom dynamic kernel loader: new kernel code is loaded into dummy container-kernel address space and executed through one of 63 kernels-stubs denoting the register count and kernel code starting address.

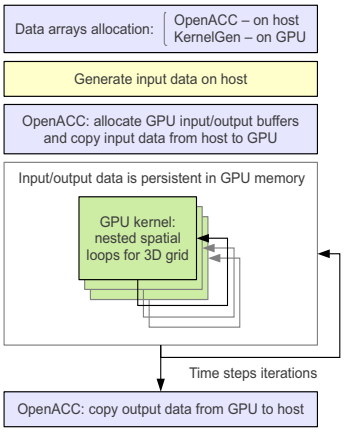


Figure 4: Organization of numerical programs in KernelGen performance test suite.

act as a container for other kernel code. Once the runtime needs to load a new kernel, its binary code is copied into the container address space, which is known through the Effective Program Counter value (LEPC instruction of Fermi ISA). This way the container can host the code of many smaller kernels one after another (with some extra offset for proper instruction cache flushing). In fact, such a dynamic kernel 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 a 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 an 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 entry point kernel performs is jump to the start of the actual kernel code specified in the fixed item of GPU constant memory (Fig. 3). As result, new kernel launches are performed without calls to *cuModuleLoad* and *cuModuleGetFunction*. Since LEPC, absolute JMP and NOP instructions are not available in CUDA C or PTX assembler, kernel loader/container, and entry points are implemented in Fermi ISA, using AsFermi assembler [16]. Entry points register counts are defined in the sections of CUBIN ELF binary and are also set by AsFermi.

C. 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, the kernel loader should either support loading of called functions or allow all kernels to reuse functions coming with the main kernel module, where they always are 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 a particular call site, which usually allows one to perform more efficient register allocation for the price of larger code size. Cloned function bodies follow the caller code and are accounted in the caller code size in CUBIN ELF records. Function addresses used in calls are hardcoded in callers assembly. This mode is activated by default;
- *cloning=no*: in this mode a single copy of the device function is shared across all call sites, possibly requiring more registers, but keeping the code very compact in comparison to cloned version. Function addresses used in calls are unresolved (*JCAL 0x0*), the ELF relocation table contains called function names for the corresponding *JCAL* instructions offsets. Shared device functions are normal functions directly visible in the CUBIN ELF symbols table or *cuobjdump*.

Some of the 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 a single module, to have all kernels in the same module
- upon main kernel load, KernelGen custom dynamic linker builds a table of function names and their ab-

Table I: 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 wave equation solver	3D	C

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

V. 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 one 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 handwritten GPU kernels, because it allows one to better analyze compiler capabilities within its class of software.

The performance test suite consists of several typical single and double precision numerical algorithms (stencils) on two-dimensional (2D) or three-dimensional (3D) regular grids. Each algorithm is performed in 2 or 3 parallel spatial loops enclosed into nonparallel time iterations loop (Fig. 4). KernelGen automatically recognizes parallel spatial loops inside nonparallel time iterations loop, while OpenACC compilers do this only with appropriate manually inserted OpenACC directives. Tests are partially adopted from [5], short descriptions are presented in Table I. The test suite is specially designed to perform comparisons with directive-based language extensions. The current version supports CUDA, OpenACC and OpenMP extensions for MIC (KernelGen compiles the same code, ignoring all directives). Fig. 6 shows normalized performance differences between test kernels compiled with CUDA (manually-parallelized), KernelGen, and three OpenACC compilers: PGI, CAPS, and PathScale (PathScale currently supports only Fermi GPUs). The corresponding absolute execution times and register counts are listed in Table II. Fig. 5 shows an extra intercomparison between KernelGen and CPU versions of the corresponding kernels.

KernelGen is compared against OpenACC kernels *manually optimized in the best possible way* using "independent", "gang", "vector" and "present" directives. The technical details and source code of performance suite is available at http://kernelgen.org/perfsuite/.

GPU kernel 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 the 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 OpenACC 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 the 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 the kernel is recompiled too often, falling back to the generic version.

Tests *matmul* and *matvec* intentionally perform an uncoalesced dot product as inner loops. In this case, compiler's best bet is to perform loop unrolling and fuse multiple loads into a 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 a factor of 3 and optimizes reduction; loads fusing is performed by *ptxas*. This optimization results in 1.5×-5.5× speedup over PGI and CAPS on K10 GPUs (Fig. 6(e)–6(h)).

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

KernelGen uses the LLVM IR module for GPU math functions, which currently demonstrates worse performance on *sincos* test in comparison to OpenACC compilers. Double-precision *sincos* does not work for some OpenACC compil-

ers.

Overall, KernelGen is on par with all commercial compilers, and almost always shows the best performance on K10 GPUs (sm_30).

VI. Conclusion

The KernelGen project implemented an original approach for automatic code porting on NVIDIA GPUs, well-suited for numerical applications. Conserving the original source code, the compiler aims to move onto the 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 a 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 lacks infrastructure for estimating kernel computational complexity and for collecting execution statistics, needed for efficient dynamic switching between CPU and GPU versions. The 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 of another kernel launch from the current kernel, without a host call.

KernelGen source code is available at the project website: http://kernelgen.org/download/.

Acknowledgments

This work has been performed on cluster "Tödi" of Swiss National Supercomputing Centre (CSCS), cluster "Lomonosov" of Moscow State University [17], compute facilities of Rutgers University, and on hardware donated by NVIDIA.

References

[1] (2011, Nov.) The OpenACC™ Application Programming Interface. version 1.0. http://www.openacc-standard.org.

[2] (2013, Apr.) OpenHMPP, new HPC open standard for many-core. http://www.caps-entreprise.com/openhmpp-directives/.

[3] M. Govett. (2009, Apr.) 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. National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.

[4] (2012, Dec.) The heterogeneous offload model for Intel® many integrated core architecture. http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf.

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

[6] 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.*, vol. 31, no. 4, p. 32, 2012.

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

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

[9] 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.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[10] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F. X. Pasquier, G. Péan, and P. Villalon, "Par4all : From convex array regions to heterogeneous computing," in *2nd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Paris, France, Jan. 2012. [Online]. Available: http://impact.gforge.inria.fr/impact2012/workshop_IMPACT/amini.pdf

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

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

[13] D. Sands. (2009, Oct.) Reimplementing llvm-gcc as a gcc plugin. http://llvm.org/devmtg/2009-10/Sands_LLVMGCCPlugin.pdf. Apple Inc. Campus, Cupertino, California.

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

[15] 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.

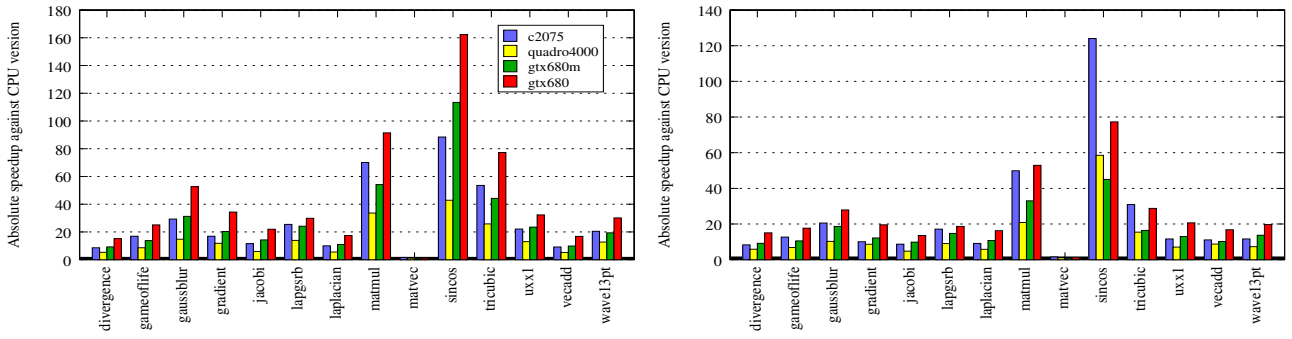
[16] Y. Hou. (2012, Dec.) AsFermi: An assembler for the NVIDIA Fermi Instruction Set. http://code.google.com/p/asfermi/.

[17] V. Voevodin, S. Zhumatiy, S. Sobolev, A. Antonov, P. Bryzgalov, D. Nikitenko, K. Stefanov, and V. Voevodin, "Practice of "Lomonosov" supercomputer," *Open Systems*, vol. 7, 2012. [Online]. Available: http://www.osp.ru/os/2012/07/13017641/

Table II: Absolute times (in microseconds) and register count of single precision test GPU kernels generated by KernelGen r1780, PGI 13.02, CAPS 3.2.4, PathScale ENZO 2013 Beta, and CUDA 5.5 on NVIDIA Tesla C2075 (Fermi sm_20), Quadro 4000 (Fermi sm_21), GTX 680 (Kepler sm_30), and GTX 680M (Kepler sm_30). Best time for each test across KernelGen, PGI, CAPS and PathScale is marked in blue. Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test.

Test	**Tesla C2075**									**GTX 680**							
	CUDA		KernelGen		PGI		CAPS		PathScale	CUDA		KernelGen		PGI		CAPS	
	time	nregs	time	nregs	time	nregs	time	nregs	time	time	nregs	time	nregs	time	nregs	time	nregs
divergence	8578	24	8463	18	8579	40	14654	22	8759	4561	28	4810	20	9662	49	11712	32
gameoflife	9126	24	10180	21	11178	28	7198	26	11088	8033	24	6867	34	9714	32	8390	25
gaussblur	12700	26	16010	56	13287	31	13146	26	9171	13884	35	8880	51	15336	36	15030	34
gradient	10370	22	10582	21	10745	42	22912	23	9555	5299	24	5233	22	9000	51	14492	29
jacobi	11073	30	7825	24	10314	23	7936	23	5524	5788	36	4105	23	5348	30	5381	26
lapgsrb	16677	35	17307	55	14387	61	17005	34	14023	13915	45	14687	40	18369	63	17612	44
laplacian	7918	24	8150	18	5071	39	6507	25	7318	4272	29	4670	21	4461	48	10564	32
matmul	690	35	994	16	718	23	2466	22	1039	621	39	548	16	668	33	1947	28
matvec	75827	17	16350	16	26357	22	85751	17	22241	64339	17	20611	16	35250	25	97733	21
sincos	15251	26	10796	22	5758	26	8202	22	4441	5915	31	5845	34	3371	29	4549	26
tricubic	46140	53	53463	63	49090	63	43981	47	133747	36729	55	36861	61	56364	63	49621	54
uxx1	18744	33	16480	32	20002	59	18898	41	18825	11908	36	11258	32	19175	63	15535	44
vecadd	5084	20	4838	12	4724	24	6068	17	4102	2626	20	2623	12	3293	31	4293	18
wave13pt	11257	24	11779	34	9886	54	23964	30	12920	7051	38	7985	34	12204	60	15622	35

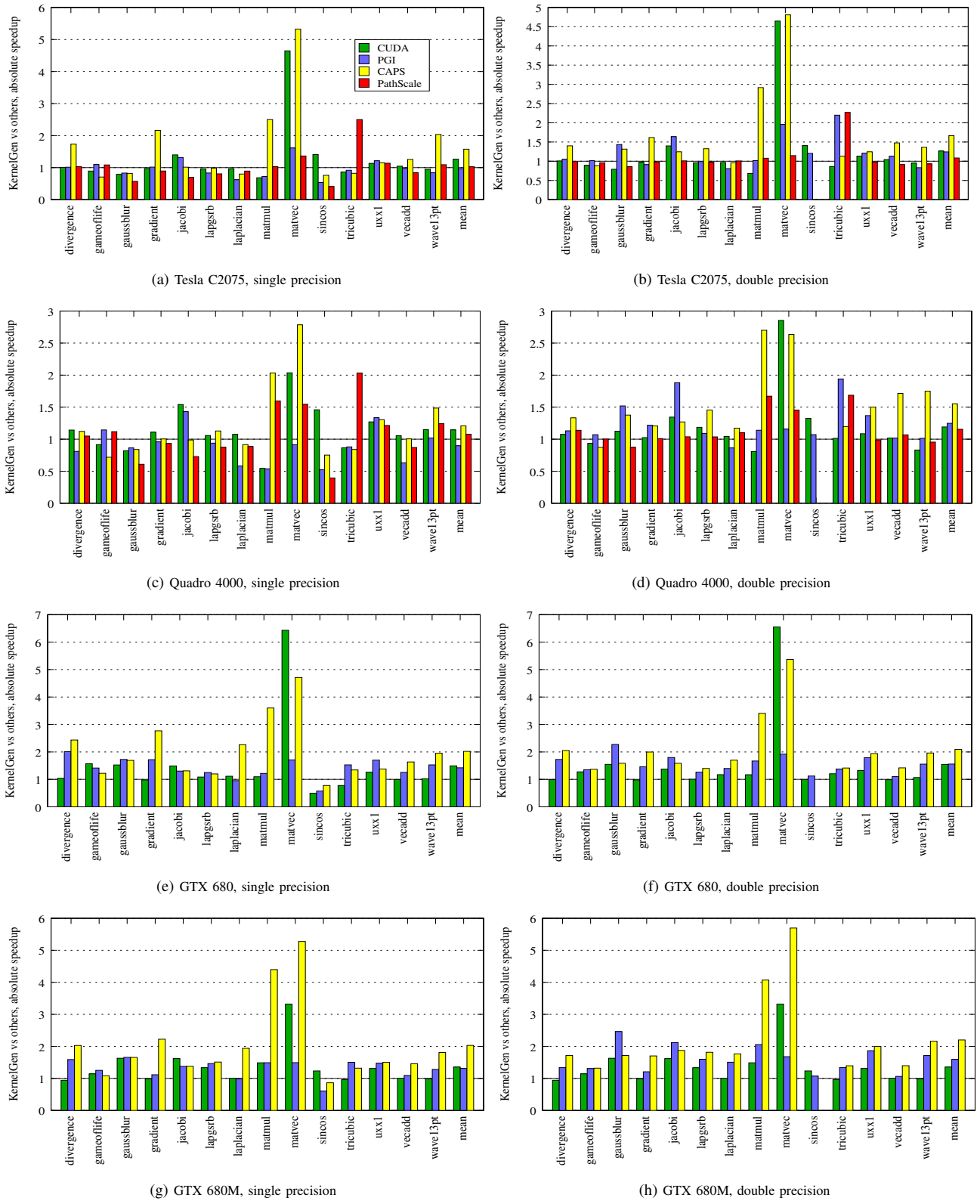
Test	**Quadro 4000**									**GTX 680M**							
	CUDA		KernelGen		PGI		CAPS		PathScale	CUDA		KernelGen		PGI		CAPS	
	time	nregs	time	nregs	time	nregs	time	nregs	time	time	nregs	time	nregs	time	nregs	time	nregs
divergence	15766	23	13810	18	11191	40	15466	22	14484	7464	28	7924	20	12595	49	16064	32
gameoflife	18435	26	20142	20	23060	28	14457	26	22514	14324	24	12555	21	15723	32	13589	25
gaussblur	25968	27	31713	56	27388	31	26562	26	19262	24407	35	14991	51	24847	36	24815	34
gradient	16828	22	15171	21	14536	42	15217	23	14158	8729	24	8855	22	9870	51	19691	29
jacobi	23369	30	15162	24	21676	23	14963	23	11047	10234	36	6325	23	8712	30	8722	26
lapgsrb	33526	34	31717	55	29754	61	35739	37	27774	24426	45	18310	40	26814	63	27654	44
laplacian	15786	24	14693	18	8560	39	13432	25	13007	7454	29	7431	21	7325	48	14407	32
matmul	1509	35	2780	14	1494	23	5653	22	2192	1089	39	731	16	1088	33	3221	28
matvec	87020	17	42747	15	39113	22	118878	17	36021	94760	17	28515	16	42524	25	150424	21
sincos	32514	29	22276	36	11623	26	16747	22	8807	10374	29	8421	22	5130	29	7263	26
tricubic	96024	51	110995	63	97652	63	93125	47	225702	62188	56	64770	61	97441	63	85468	54
uxx1	35612	40	28040	33	37450	59	36532	41	33995	20337	36	15521	32	22862	63	23283	44
vecadd	8897	15	8435	12	5331	24	8467	17	7341	4480	20	4455	12	4869	31	6491	18
wave13pt	21830	25	19013	34	1939	54	28230	30	23578	12297	38	12462	34	15987	60	22544	35



(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 (single core, optimization flag used: -O3). Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test.



(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, PathScale ENZO 2013 Beta, and CUDA 5.5 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.