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

Dmitry Mikushin
Institute of Computational Science,
Faculty of Informatics,
University of Lugano,
Switzerland
Email: dmitry@kernelgen.org

Nikolay Likhogrud
Faculty of Computational
Mathematics and Cybernetics,
Lomonosov Moscow State University,
Russia
Email: n.lihogrud@gmail.com

Eddy Z. Zhang
Department of Computer Science,
Rutgers University,
New Brunswick, New Jersey
Email: eddy.zhengzhang@cs.rutgers.edu

***Abstract*—GPUs are getting more and more important in scientific computing, slowly growing from peripheral accelerators into central processing nodes. In the same time, current OpenACC and HMPP directive-based approaches imply offloading of “hot” code regions onto GPUs, as the only one possible development strategy. Furthermore, as the portion of GPU code and target application size grows, essential limitations of this programming model such as unsupported external functions calls and insufficient data dependencies analysis become critical. This paper introduces more robust parallelism detection approach based on runtime-assisted polyhedral analysis and LLVM compiler infrastructure, as well as a novel GPU-centric execution model. Complemented by GCC frontend, NVPTX backend and other parts, complete GPU compiler prototype called KernelGen has been developed. Test kernels produced by KernelGen are up to 60% faster than PGI OpenACC compiler. KernelGen is proposed as community open-source platform for GPU compiler optimizations research.**

I. Introduction

The mass use of the GPU-enabled computing clusters requires broad adaptation of many complex applications. The CUDA and OpenCL programming models are well suited for small programs with several concentrated computational kernels. However, for larger applications consisting of many individual interacting blocks, such as numerical models, the complexity of setting up efficient interaction between original code and new GPU code portions increases dramatically. Many companies and research groups still postpone porting of their applications, since in addition to GPU-specific implementation, it is often required to keep back the conservative CPU version, which eventually results into larger resources consumptions on development and support of segmented code base. Moreover, scientific specialists accustomed to work with simple CPU code in Fortran and focus on their problem domain, could hardly be forced to deal with complicated details of GPU parallelism, which limits further research activities in accelerated codes. Driven by the need to simplify this process, a number of new types of programming models have emerged:

- **The directive-based extensions to existing high-level languages with user-controlled parallelism, similar to OpenMP.** This type of programming technologies introduce sets of directives (annotations) for marking up the code regions intended for execution on GPU. Based on this extra information, compiler automatically generates hybrid executable binary. In order to standardize the set of directives for C/C++/Fortran, commercial compiler vendors formed OpenACC [7] and OpenHMPP [8] consortiums. F2C-ACC source-to-source processor is able to perform directive-based GPU code generation for a subset of Fortran programming language [10]. Similar set of directives is being developed by Intel for Many Integrated Core (MIC) platform [9].
  Despite the higher code porting robustness, 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 settings consistency, others – blindly follow the user preference. Compilers are often “too careful” while making decisions about loops parallelism based only on internal analysis, and user may need to force porting with additional directives. Most of directive-based compilers do not support generation of GPU kernels for loops with calls to functions from other compilation units or libraries, introducing significant limitation for this type of methods in large applications.
- **The domain-specific languages (DSL) specially designed to express the parallelism of algorithms in specific problem domain.** In recent years many DSLs and embedded DSLs have emerged. Their main idea is to bring closer the language features and requirements of programs in specific problem domain, and in the same time – to eliminate too strict specialization for particular hardware. In fact, DSLs introduce an additional level of programming abstraction, which is lowered by compiler or source-to-source processor into code for specific target architectures. For instance, PATUS [1] 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 [5] is proposed for the similar problem domain, but embeds into C++ and heavily uses templates. Binaries could be generated for CPUs and NVIDIA GPUs. An embedded DSL called Halide [6] supports code generation for x86-64/SSE, ARM v7/NEON and NVIDIA GPUs, and is intended mainly for image processing.

The testing of DSLs/eDSLs is usually performed in comparison to hand-tuned programs, making it hard to evaluate the possible benefits over directive-based compilers and other DSLs. Every unique DSL typically has only one developer group, limiting the concurrency. Utilizing DSLs in existing programs usually require massive code rewrite, resulting into the similar drawbacks as for CUDA and OpenCL discussed earlier.

- **Automatic analysis for code parallelism based on heuristics or polyhedral analysis.** Technologies of this type are intended for computing data dependencies and 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. Similar solution capable of transforming C/C++ loops into CUDA kernels called PPCG has been developed for Clang [12]. Both projects utilize CLooG polyhedral analysis library [13] for identifying parallel loops in compler internal representation (IR). Source-to-source compiler Par4all [14] transforms C and Fortran code into CUDA, OpenCL or OpenMP kernels using another polyhedral analysis system called PIPS.

In any case, explicit CUDA, directive-based programming models and specialized languages still presume significant manual code modification. For this reason, it is practically very difficult to completely port a large enough application on the GPU. And 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 model with the PGI Accelerator, the time spent on data synchronization is 40-60 % of the total time [16].

Comparing existing technologies capabilities with the typical scenarios and issues of large applications porting, a number of desirable properties of next-generation compiler frameworks could be derived:

- Support a large set of *existing* popular programming languages.;
- Automatically analyze the source code for parallelism and transform it into GPU kernels, without developer assistance;
- Code generation process, fully compatible with resular host code compilation;
- Minimize the dataflow between the main system memory and the GPU;
- Co-exist with other levels of parallelism, first of all – with MPI.

The goal of KernelGen project is to create a compiler, fitting these requirements and developing a roadmap for improving the necessary compiler technologies. Clearly, such system cannot be built neither as a directive-based compiler, nor as a DSL, however, it could use some existing research tools for automatic loops analysis. KerelGen is being developed at the junction of numerical methods, compiler theory and practical implementation of the new technology.

This article proposes a prototype KernelGen GPU compiler, aiming to perform original program code compilation into mixed CPU+GPU binary, and to organize seamless interoperation of host and device parts. Section II describes compiler pipeline, linking, execution model and memory management. Section III explains the necessary modifications made to existing parallel loops analysis and transformation tool, in order to generate GPU kernels. Sections IV and V are dedicated to auxiliary compiler subsystems and performance testing, respectively.

II. KernelGen compiler pipeline

During compiler toolchain development, it is important to choose the most suitable existing infrastructure, by number of criteria: 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 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 [15] 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.

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. Although compatibility is extremely important to support large complex applications, it is often neglected in the novel programming models design. 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 some simple serial code and 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 [17] 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.

A. 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(unsigned 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).

B. 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.

C. 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.

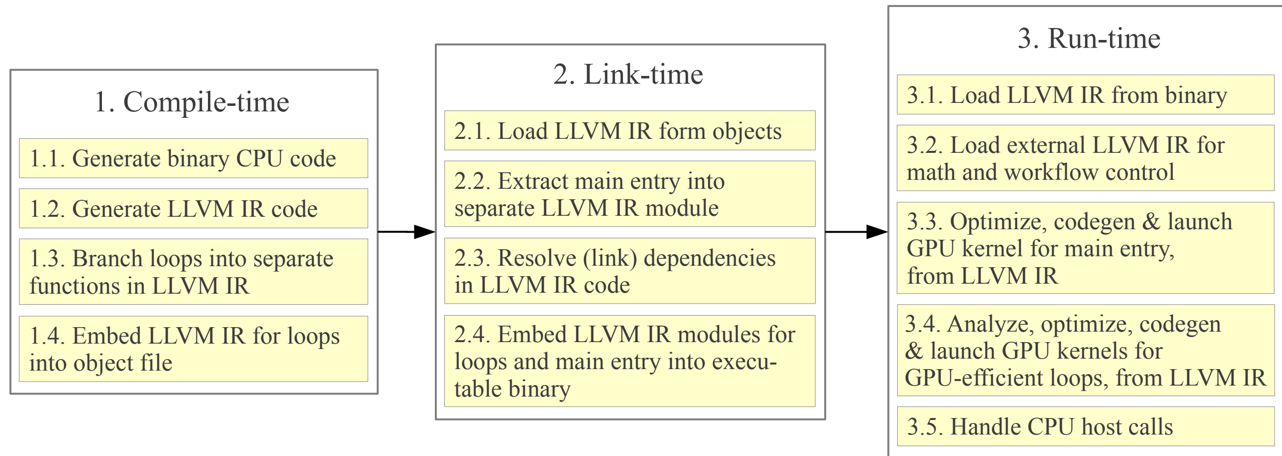

Fig. 1. KernelGen compiler pipeline

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.

D. 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 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.

III. Generating GPU kernels for parallel loops

Polly [3] (from the polyhedral analysis), a part of LLVM infrastructure, is a set of loops transformation passes based on CLooG [13]. 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

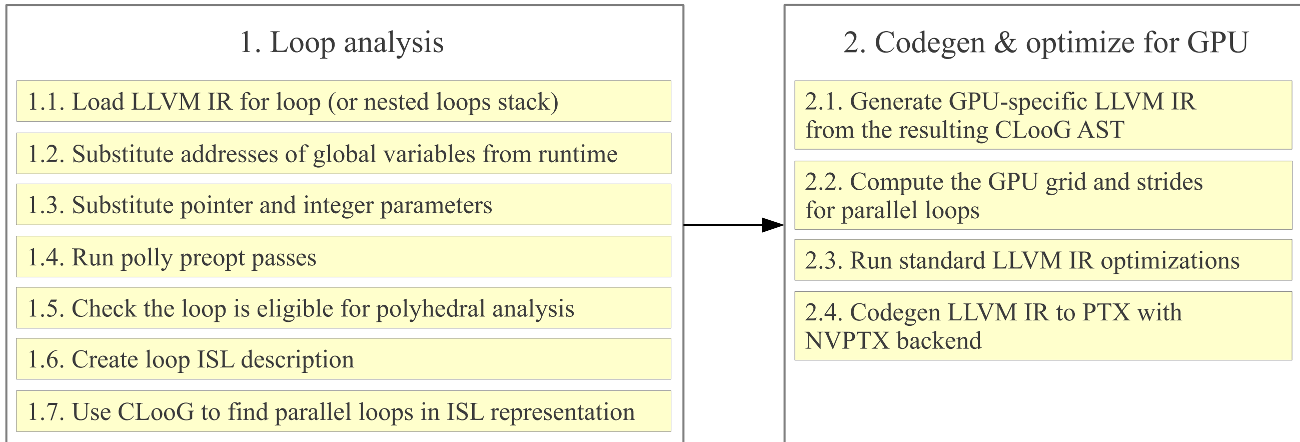

Fig. 2. Loops analysis and parallel GPU code generation pipeline in KernelGen compiler. Optimization is performed for entire SCoP, code generation – for each individual function

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 *while*-loops, which is not supported by 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 used. In OpenMP case, if the outer loops 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) Map loop iteration space is manually mapped onto GPU compute grid, favoring coalesced GPU global memory transactions;
2) Not only the outer loop, but all nested loops are processed recursively, to utilize the multidimensional grids of threads.

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.

A. *Optimizing GPU kernels for memory locality*

TODO

IV. Extra runtime subsystems

A. *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 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.

B. *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 this

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 and the specified GPU compute grid is lauched. 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. 3). 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 [4]. 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, 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.

D. 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.

V. Testing

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 analyze compiler capabilities within its class of systems.

Performance test suite consists of several typical single-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. Tests are partially adopted from [2], detailed descriptions and source codes are presented in [20]. The test suite is specially designed to perform comparisons with directive-based language

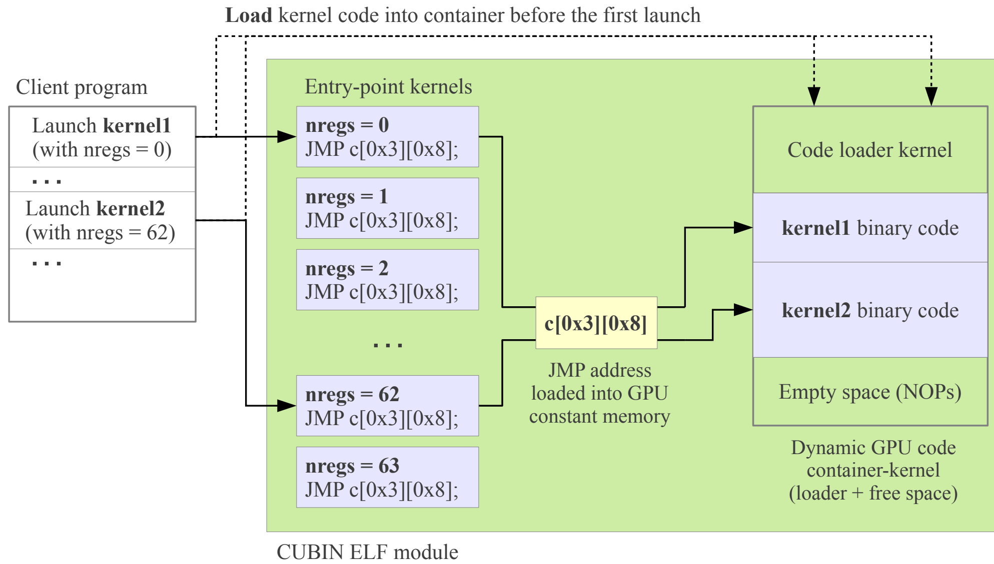

Fig. 3. KernelGen custom dynamic kernels loader: new kernels code is loaded into dummy container-kernel address space and executed through one of 63 kernels-stubs denoting the register count and kernel code starting address.

extensions. Current version supports OpenACC and OpenMP extensions for MIC. Fig. 4 shows normalized performance differences between tests kernels compiled with KernelGen and PGI OpenACC. The corresponding absolute execution times and register counts are listed in Table I. Tests *jacobi*, *matmul* and *sincos* are implemented in Fortran, the rest of test suite – in C. KernelGen automatically recognizes parallel spatial loops inside non-parallel time iterations loop, while PGI does this only with appropriate manually inserted OpenACC directives.

Additional testing on COSMO [18] and WRF [19] numerical models showed KernelGen is able to generate consistent GPU-enabled executables for complex applications in reasonable time.

VI. Conclusion

KernelGen project implemented an original approach for automatic code porting on NVIDIA GPUs, well-suited for complex 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 commercial PGI OpenACC compiler.

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 (HP2C, hp2c.ch), testing is performed on cluster “Lomonosov” of Lomonosov Moscow state University [21], cluster “Tödi” of Swiss National Supercomputing Centre (CSCS), and on hardware donated by NVIDIA and HP.

References

[1] M. Christen, “Generating and Auto-Tuning Parallel Stencil Codes,” Ph.D. dissertation, University of Basel, Switzerland, 2011.

[2] M. Christen, O. Schenk, Y. Cui, PATUS for Convenient High-Performance Stencils: Evaluation in Earthquake Simulations. SC ’12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.

[3] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, L.-N. Pouchet, “Polly – Polyhedral Optimization in LLVM,” *IMPACT 2011 (at CGO 2011), Charmonix France*, April 2011.

[4] Y. Hou et al, “AsFermi: An assembler for the NVIDIA Fermi Instruction Set,” http://code.google.com/p/asfermi/ (03.12.2012).

[5] T. Gysi, “HP2C Dycore,” Presentation, http://mail.cosmo-model.org/pipermail/pompa/attachments/20120306/079fadc1/DWD_HP2C_Dycore_120305.pdf, *Workshop on COSMO dynamical core rewrite and HP2C project*, March 2012, Offenbach, Germany.

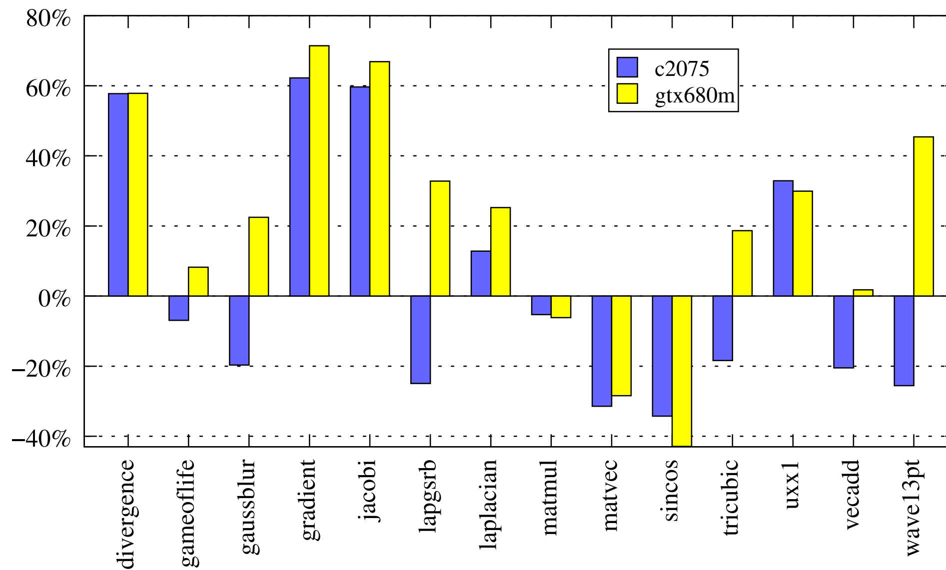

Fig. 4. Comparing performance of test GPU kernels generated by KernelGen r1578 and PGI 12.10 on NVIDIA Tesla C2075 (Fermi sm_20) and GTX 680M (Kepler sm_30). Positive values – KernelGen version is faster than PGI, negative values – PGI version is faster than KernelGen. Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

TABLE I
COMPARING EXECUTION TIME (SEC) AND REGISTER COUNT (NREGS) OF TEST GPU KERNELS GENERATED BY KERNELGEN R1578 AND PGI 12.10 ON NVIDIA TESLA C2075 (FERMI SM_20) AND GTX 680M (KEPLER SM_30). MEASUREMENTS ARE AVERAGED FROM 10 INVOCATIONS OF ALL TESTS AND 10 ITERATIONS INSIDE EVERY TEST

	NVIDIA Tesla C2075				NVIDIA GTX 680M			
	KernelGen		PGI		KernelGen		PGI	
	time	nregs	time	nregs	time	nregs	time	nregs
divergence	0.010920	18	0.017224	36	0.009811	20	0.015487	48
gameoflife	0.011383	21	0.010597	27	0.014631	21	0.015831	27
gaussblur	0.016835	56	0.013521	34	0.020240	51	0.024789	40
gradient	0.012687	21	0.020579	35	0.009314	22	0.015964	47
jacobi	0.009008	24	0.014380	26	0.007355	23	0.012274	31
lapgsrb	0.034975	55	0.026247	63	0.019294	40	0.025616	63
laplacian	0.009970	18	0.011246	32	0.008415	22	0.010537	48
matmul	0.001514	13	0.001434	33	0.001631	14	0.001531	37
matvec	0.049047	12	0.033639	27	0.062920	16	0.045024	27
sincos	0.012112	22	0.007962	25	0.009351	22	0.005341	29
tricubic	0.074085	60	0.060450	63	0.086248	61	0.102335	63
uxx1	0.024248	32	0.032225	53	0.019377	32	0.025174	62
vecadd	0.006269	12	0.004983	28	0.005046	12	0.005135	36
wave13pt	0.025364	34	0.018885	63	0.013564	34	0.019723	63

[6] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, F. Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines, SIGGRAPH 2012.

[7] The OpenACC^{TM}Application Programming Interface. Version 1.0, November, 2011, http://www.openacc-standard.org (03.12.2012).

[8] OpenHMPP, New HPC Open Standard for Many-Core, http://www.openhmpp.org/en/OpenHMPPConsortium.aspx (03.12.2012).

[9] The Heterogeneous Offload Model for Intel® Many Integrated Core Architecture, http://software.intel.com/sites/default/files/article/326701/heterogeneous-programming-model.pdf (03.12.2012).

[10] M. Govett, "Development and Use of a Fortran → CUDA translator to run a NOAA Global Weather Model on a GPU cluster," *Path to Petascale: Adapting GEO/CHEM/ASTRO Applications for Accelerators and Accelerator Clusters*, Presentation, http://gladiator.ncsa.uiuc.edu/PDFs/accelerators/day2/session3/govett.pdf, April 2009, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.

[11] A. Kravets, A. Monakov, A. Belevantsev, "GRAPHITE-OpenCL: Automatic parallelization of some loops in polyhedra representation," *GCC Developers' Summit,* October 2010, Ottawa, Canada.

[12] S. Verdoolaege et al, "PPCG – C to CUDA processor," http://repo.or.cz/w/ppcg.git (03.12.2012).

[13] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, September, 2004, Juan-les-Pins, France.

[14] M. Torquati, M. Venneschi, M. Amini, S. Guelton, R. Keryell, V. Lanore, F.-X. Pasquier, M. Barreteau, R. Barrère, C.-T. Petrisor, . Lenormand, C. Cantini, F. De Stefani. An innovative compilation tool-chain for embedded

multi-core architectures. Embedded World Conference 2012. Nuremberg, Germany, 2/2012.

[15] D. Sands, “Reimplementing llvm-gcc as a gcc plugin,” *Third Annual LLVM Developers’ Meeting,* Presentation, http://llvm.org/devmtg/2009-10/Sands_LLVMGCCPlugin.pdf, October 2009, Apple Inc. Campus, Cupertino, California.

[16] M. Wolfe, C. Toepfer, “The PGI Accelerator Programming Model on NVIDIA GPUs Part 3: Porting WRF,” http://www.pgroup.com/lit/articles/insider/v1n3a1.htm (03.12.2012).

[17] J. Squyres, G. Bosilca, S. Sumimoto, R. vandeVaart, “Open MPI State of the Union,” *Open MPI Community Meeting*, Presentation, http://www.open-mpi.org/papers/sc-2011/Open-MPI-SC11-BOF-1up.pdf, Supercomputing 2011.

[18] Consortium for Small-scale Modeling, http://www.cosmo-model.org/ (03.12.2012).

[19] The Weather Research & Forecasting Model, http://www.wrf-model.org/index.php (03.12.2012).

[20] KernelGen Performance Test Suite, https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite (27.01.2013).

[21] Voevodin Vl.V., Zhumatiy S.A., Sobolev S.I., Antonov A.S., Bryzgalov P.A., Nikitenko D.A., Stefanov K.S., Voevodin Vad.V. Practice of ”Lomonosov” Supercomputer // Open Systems J. - Moscow: Open Systems Publ., 2012, no.7. [http://www.osp.ru/os/2012/07/13017641/] (In Russian).