Università della Svizzera italiana

Facoltà di scienze informatiche

Institute of Computational Science

Evaluating manual and compiler-driven parallelization of stencil micro-applications on a GPU-enabled cluster

Dmitry Mikushin, Olaf Schenk

February 21, 2014

SIAM Conference on
Parallel Processing
for Scientific Computing

February 18-21, 2014
Marriott Portland
Downtown Waterfront
Portland, Oregon, USA

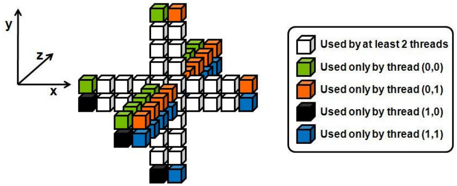
Annotation

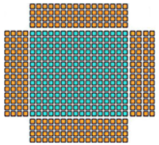
In this talk we will demonstrate how parallelization and further optimization of stencil codes for GPUs could be automated by compiler toolchains. By example of wave equation stencil, hand-written naive and optimized for locality versions will be compared against compiler-generated parallel code, presenting the roofline performance, efficiency of tiling, JIT-compilation and other properties. The results of benchmarking KernelGen and PPCG auto-parallelizing compilers as well as one commercial OpenACC compiler will be presented on a set of 10 stencil micro-applications.

3D Finite Difference Computation on GPUs using CUDA

A famous paper by Paulius Micikevicius, **280** citations:

- Stencil for 3D wave equation (6-12th order in space, 2nd order in time)
- 2D slices are tiled in GPU shared memory
- Columns of 3rd dimension are cached in GPU thread registers
- Released in 2009, performance tested on Tesla S1060 (GT200)


y
z
x
Used by at least 2 threads
Used only by thread (0,0)
Used only by thread (0,1)
Used only by thread (1,0)
Used only by thread (1,1)



Questions we addressed in this study

1. Are those frequently cited tiling optimizations still beneficial on modern GPUs?
2. Are there any new stencil-related optimizations yet to discover?
3. To what extent the generation of GPU kernels for stencils could be automated?
    - Can compilers generate *efficient* parallel GPU code for stencils?
    - What manual optimizations could be easily implemented by compiler?

Example wave 3D stencil

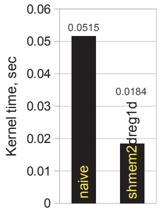
```
for (int k = 2; k < ns - 2; k++)
    for (int j = 2; j < ny - 2; j++)
        for (int i = 2; i < nx - 2; i++)
        {
                w2[k][j][i] =  m0 * w1[k][j][i] - w0[k][j][i] +

                    m1 * (
                        w1[k][j][i+1] + w1[k][j][i-1]   +
                        w1[k][j+1][i] + w1[k][j-1][i]   +
                        w1[k+1][j][i] + w1[k-1][j][i]) +
                    m2 * (
                        w1[k][j][i+2] + w1[k][j][i-2]   +
                        w1[k][j+2][i] + w1[k][j-2][i]   +
                        w1[k+2][j][i] + w1[k-2][j][i]);
}
```

input: *w0* (1 pt/iter), *w1* (13 pt/iter)

output: *w2* (1 pt/iter)

Reproducing P. Micikevicius method for wave13pt on S1070

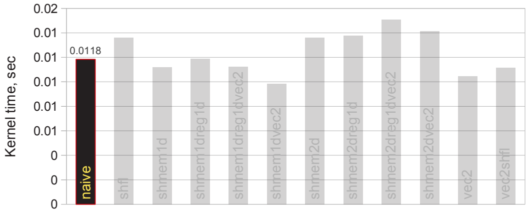
- $\{32, 16, 1\}$ blocks, maxrregcount=32
- each thread is handling points of the same vertical column
- array items reusable by the next point are tiled in registers
- *naive* – naïve CUDA version, *shmem2dreg1d* – with above optmzns applied
- *shmem2dreg1d* is almost **3**$\times$ faster than *naive* on S1070 – remember this result!

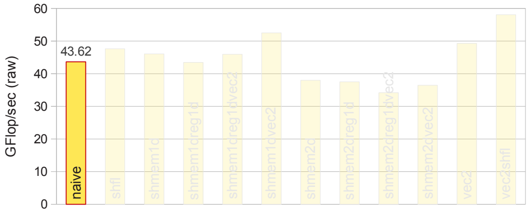


`wave13pt` on Tesla S1070 (GT200/SM_13), single precision

Naïve CUDA implementation

- $\{128, 1, 1\}$ blocks
- one grid point per thread



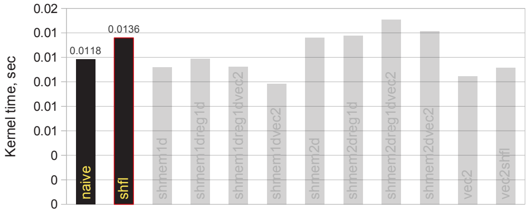


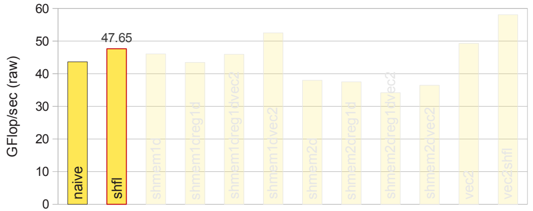
`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Warp data shared with *shuffle* instruction

- $\{128, 1, 1\}$ blocks
- maxrregcount=32
- load array value, use and pass to neighboring thread
- shuffles are accounted into FLOPS, hence more FLOPS, but larger kernel time
- needs shareable flops to be beneficial

```
float val = w1[k][j][i+2];
float result = m2 * val;
val = __shfl_up(val, 1);
if (laneid == 0) val = w1[k][j][i+1];
```

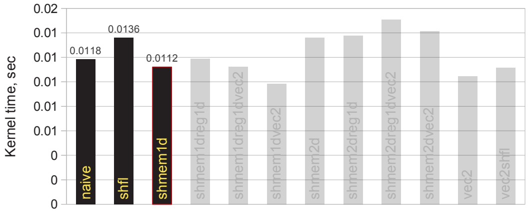


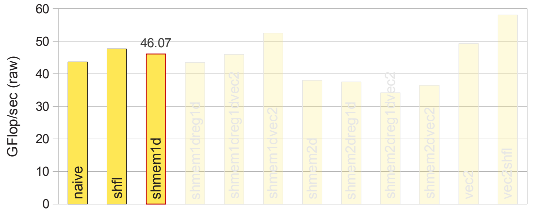


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Tiling 1D line of *w1* array in shmem

- $\{128, 1, 1\}$ blocks
- maxrregcount=32

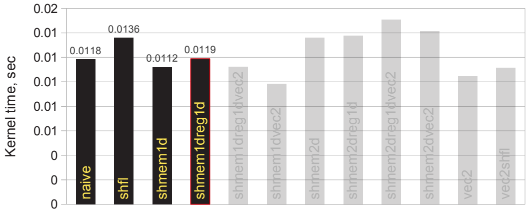


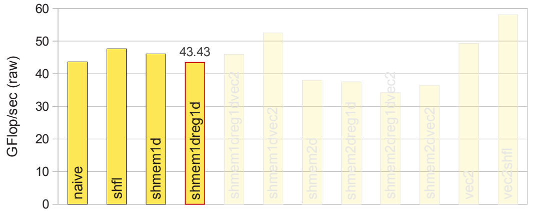


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Tiling 1D line of *w1* array in shmem, vertical line in registers

- $\{128, 1, 1\}$ blocks
- maxrregcount=32
- each thread is handling points of the same vertical column
- array items reusable by the next point are tiled in registers

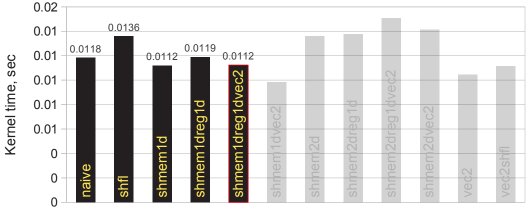


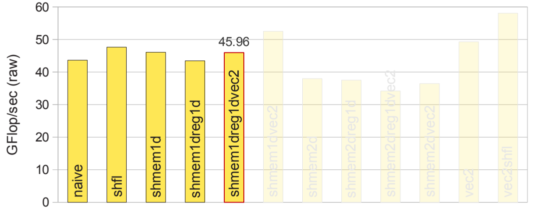


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

1D line of *w1* array in shmem, vertical line in regs, vector LD/ST

- $\{128, 1, 1\}$ blocks
- maxrregcount=32
- each thread is handling points of the same vertical column
- array items reusable by the next point are tiled in registers
- each thread handles two points, using 2-element vector load/stores (LD.64/ST.64)

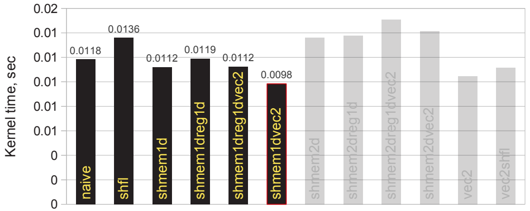


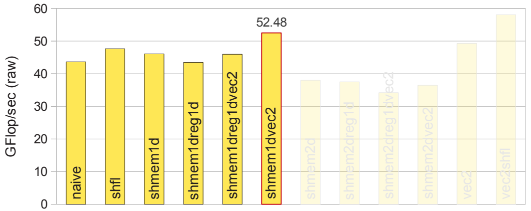


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

1D line of *w1* array in shmem, vector LD/ST

- $\{128, 1, 1\}$ blocks
- maxrregcount=32
- each thread handles two points, using 2-element vector load/stores (LD.64/ST.64)

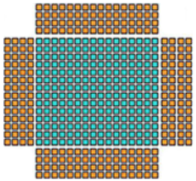


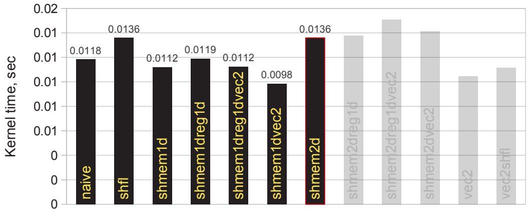


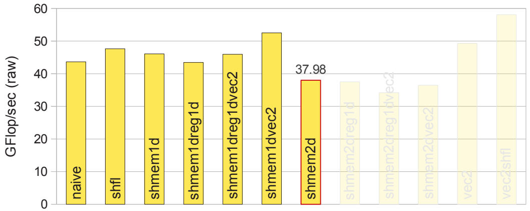
`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Tiling 2D slice of *w1* array in shmem

- $\{32, 16, 1\}$ blocks
- maxrregcount=32
- additional time is spent on loading of shadow boundaries by a subset of threads



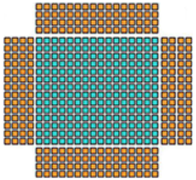


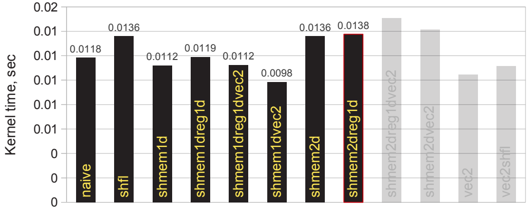


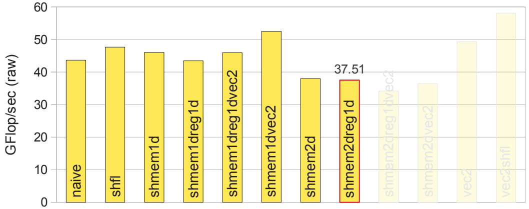
`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Tiling 2D slice of *w1* array in shmem, vertical line in registers

- $\{32, 16, 1\}$ blocks
- maxrregcount=32
- array items reusable by the next point are tiled in registers



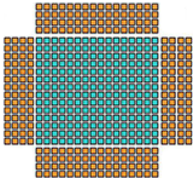


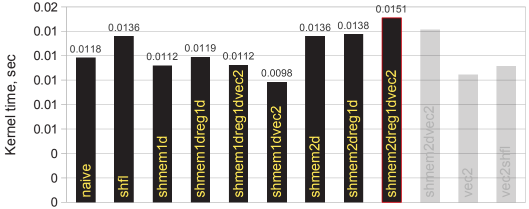


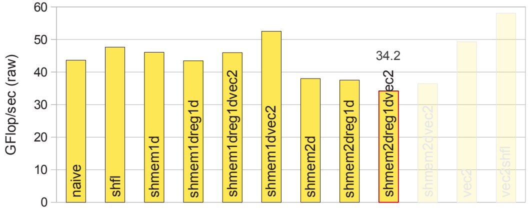
`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

2D slice of *w1* array in shmem, vertical line in regs, vector LD/ST

- $\{32, 16, 1\}$ blocks
- maxrregcount=32
- array items reusable by the next point are tiled in registers
- each thread handles two points, using 2-element vector load/stores



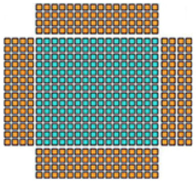


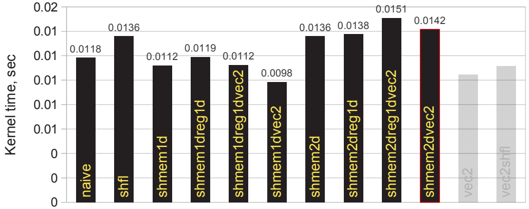


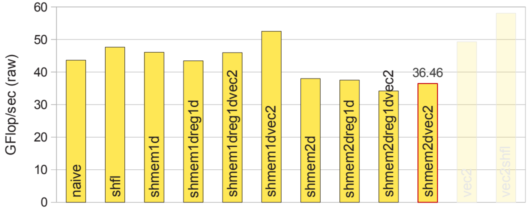
`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

2D slice of *w1* array in shmem, vector LD/ST

- $\{32, 16, 1\}$ blocks
- maxrregcount=32
- each thread handles two points, using 2-element vector load/stores (LD.64/ST.64)



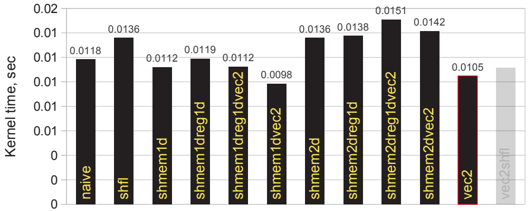


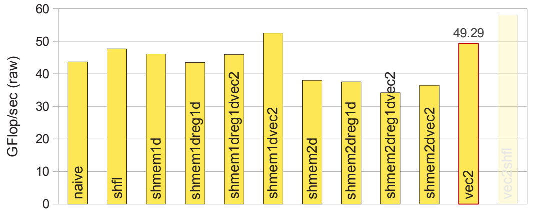


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Vectorized load/store

- $\{128, 1, 1\}$ blocks
- each thread handles two points, using 2-element vector load/stores (LD.64/ST.64)

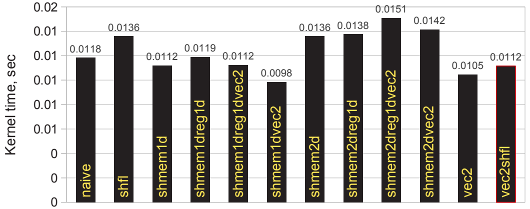


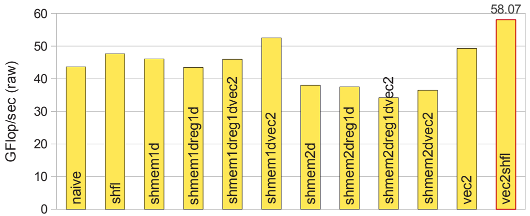


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

Vectorized load/store, warp data shared with *shuffle*

- $\{128, 1, 1\}$ blocks
- each thread handles two points, using 2-element vector load/stores (LD.64/ST.64)
- load array value, use and pass to neighboring thread
- shuffles are accounted into FLOPS, hence more FLOPS, but larger kernel time
- needs shareable flops to be beneficial

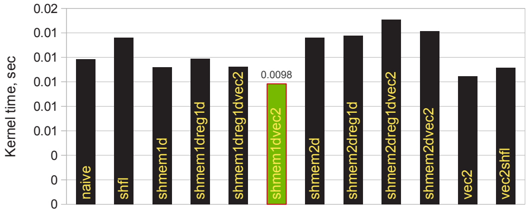


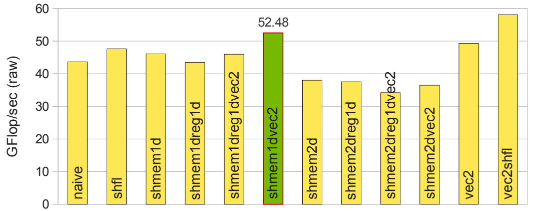


`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

The best manually-optimized version

- 1D shared memory with vectorization is the best on GTX 680M
- the contribution of shmem is minor
- vectorization-only is the best on Tesla K20 (Piz Daint)
- compared to 3× improvement on S1070, here tiling shows almost no speedup





`wave13pt` on GeForce GTX 680M (GK104/SM_30), single precision

What is the effect of vectorization at the low-level?

- Surprisingly, the code of scalar kernel for memory-bound stencil appears to be compute-bound
- 2-element vectorization improves the memory efficiency and reduces arithmetics (less indexing?)

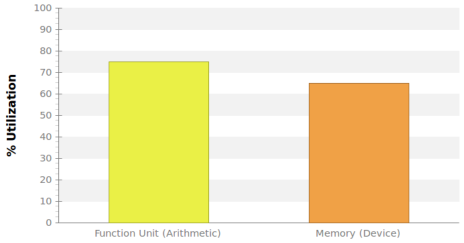


Figure : CUDA naïve kernel

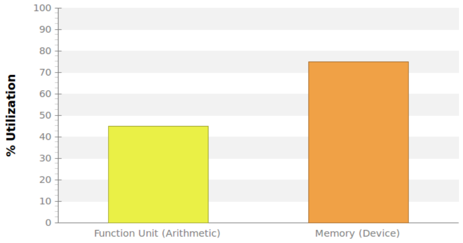
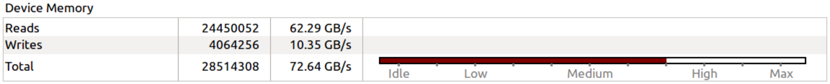


Figure : CUDA vectorized kernel

What is the effect of vectorization at the low-level?

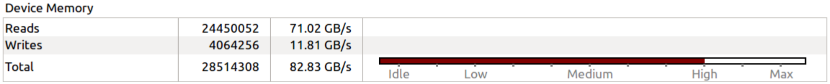
- Global memory throughput is higher in vectorized version (very close to cuMemcpyDtoD, which is 84 Gb/sec on this device).



Device Memory

Reads	24450052	62.29 GB/s	
Writes	4064256	10.35 GB/s	
Total	28514308	72.64 GB/s	

Figure : CUDA naïve kernel



Device Memory

Reads	24450052	71.02 GB/s	
Writes	4064256	11.81 GB/s	
Total	28514308	82.83 GB/s	

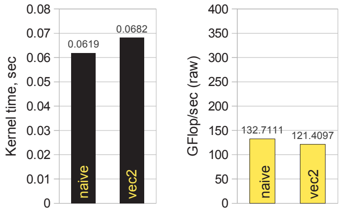
Figure : CUDA vectorized kernel

What is the effect of vectorization at the low-level?

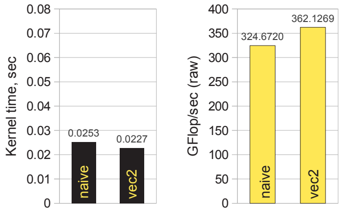
- kernel enjoys 4% higher global memory load efficiency
- 13% higher global store/write, 13% higher global read and 7% lower load throughput (less reloads to cache**?**)
- 13% higher L2 write, 7% lower L2 read, 7% lower L2 read from L1 throughput
- 40% less used issue slots
- 9% less control-flow, 14% less load-store instructions
- 34% higher instruction, 2.83$\times$ higher global memory replay overhead
- 19% less issued IPC, 45% less instructions per warp
- 58032 global stores, 84% more global stores replayed due to divergence (**??**)
- requires 1.5-2$\times$ more registers per thread and may show no speedup in event of excessive spilling

Another vectorization test on K20: tricubic

- Unlike `wave13pt`, `tricubic` test is compute-bound and has a large register footprint
- No effect of vectorization on GTX 680M due to excessive register spilling
- However, the same 12% speedup is on Tesla K20, where the larger register file is available



`tricubic` on GeForce GTX 680M (GK104/SM_30), single precision



`tricubic` on Tesla K20 (GK110/SM_35), single precision

Extra minor observations

- 13% speedup with alignment of boundary threads:

```
for (int k = 2+ k_start; k < ns — 2; k += k_inc)
    for (int j = 2 + j_start; j < ny — 2; j += j_inc)
        for (int i = 2 + i_start; i < nx — 2; i += i_inc)
        {
            ...
        }
```

$\Rightarrow$

```
for (int k = 2+ k_start; k < ns — 2; k += k_inc)
    for (int j = 2 + j_start; j < ny — 2; j += j_inc)
        for (int i = i_start; i < nx — 2; i += i_inc)
        {
            if (i < 2) continue;
        }
```

- Caching in texture memory is enabled by default for noalias pointers (LDG instead of LD) on GK110. However, manual disabling of texture caching does not affect the perf.
  $\Rightarrow$ Is it possible to create a tiling strategy making the texture cache beneficial for stencils?

Auto-parallelizing compilers

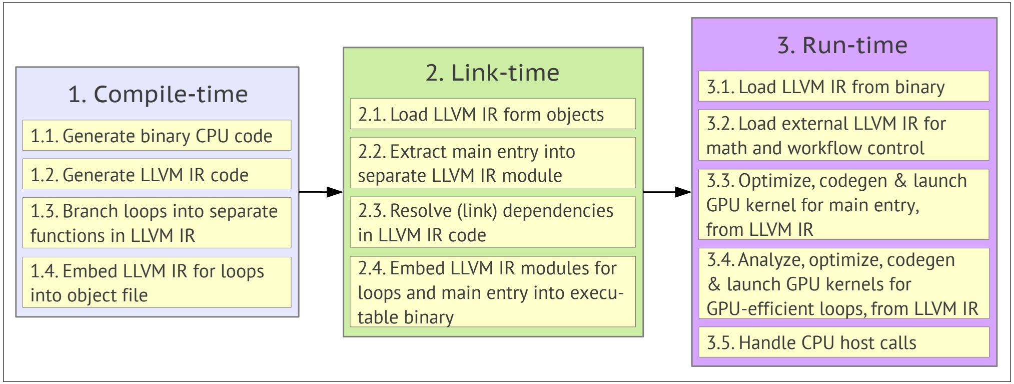
- OpenACC
  - Require a lot of manual assistance and introduce many practical limitations
- Polyhedral toolchains
  - **PPCG** – the newest polyhedral tool
    - Transforms loops into parallel loops for execution on GPU
    - Still requires manual tuning, without tuning is **10**× slower than naïve CUDA
    - Clang frontend, then – source-to-source
  - **KernelGen** – a conservative descendant of LLVM Polly
    - Checks loops for parallelism, assisted by runtime info and executes parallel loops on GPU
    - Performs substitution of the runtime constants, reducing the register footprint at the price of JIT
    - Default heuristics allow competitive performance, no tuning is needed
    - GCC frontend, LLVM backend

KernelGen dependencies

- **GCC** – for frontends and regular compiling pipeline (GPL)
- **DragonEgg** – GCC plugin for converting GCC's IR (gimple) into LLVM IR (GPL)
- **LLVM** – for internal compiler infrastructure (BSD)
- **Polly** – for loops parallelism analysis (BSD+GPL)
- **NVPTX backend** – for emitting LLVM IR into PTX/GPU intermediate assembly (BSD)
- **PTXAS** – for emitting PTX/GPU into target GPU ISA (proprietary, no source code)
- **AsFermi** – for necessary CUBIN-level tricks in Fermi GPU ISA (MIT/BSD)
- **NVIDIA GPU driver** – for deploying the resulting code on GPUs (proprietary, no source code)
- **CUDA-aware MVAPICH2** – for more efficient GPU peer-to-peer communication in KernelGen-MPI programs (BSD)

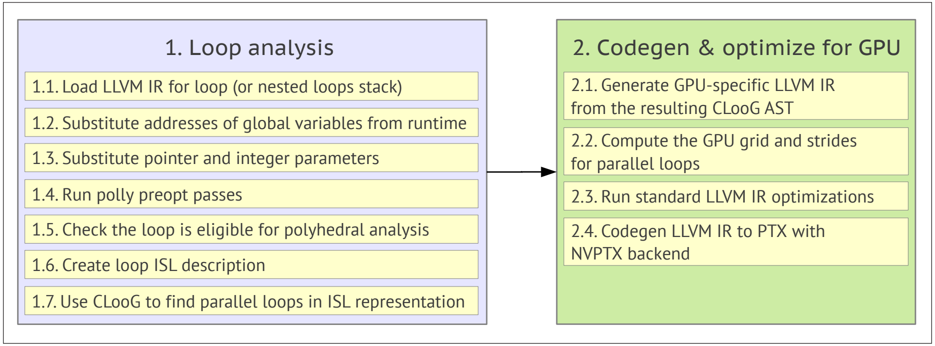
KernelGen compiler pipeline

KernelGen conserves original host compiler pipeline (based on GCC), extending it with parallel LLVM-based pipeline, which is activated and/or used if specific environment variables are set.


1. Compile-time
1.1. Generate binary CPU code
1.2. Generate LLVM IR code
1.3. Branch loops into separate functions in LLVM IR
1.4. Embed LLVM IR for loops into object file
2. Link-time
2.1. Load LLVM IR form objects
2.2. Extract main entry into separate LLVM IR module
2.3. Resolve (link) dependencies in LLVM IR code
2.4. Embed LLVM IR modules for loops and main entry into executable binary
3. Run-time
3.1. Load LLVM IR from binary
3.2. Load external LLVM IR for math and workflow control
3.3. Optimize, codegen & launch GPU kernel for main entry, from LLVM IR
3.4. Analyze, optimize, codegen & launch GPU kernels for GPU-efficient loops, from LLVM IR
3.5. Handle CPU host calls

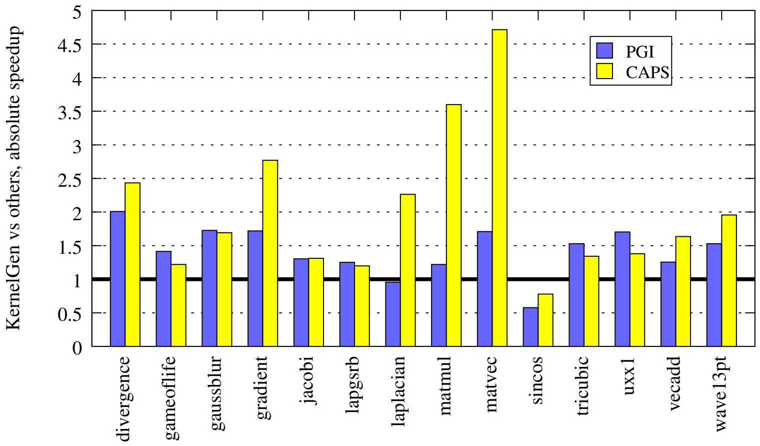
KernelGen loops analysis pipeline

KernelGen takes part of loop analysis into runtime, in order to process only really used loops, and do it better with help of additional information available from the execution context. Introduced runtime overhead is neglectible, if the loop is invoked frequently.


1. Loop analysis
1.1. Load LLVM IR for loop (or nested loops stack)
1.2. Substitute addresses of global variables from runtime
1.3. Substitute pointer and integer parameters
1.4. Run polly preopt passes
1.5. Check the loop is eligible for polyhedral analysis
1.6. Create loop ISL description
1.7. Use CLooG to find parallel loops in ISL representation
2. Codegen & optimize for GPU
2.1. Generate GPU-specific LLVM IR from the resulting CLooG AST
2.2. Compute the GPU grid and strides for parallel loops
2.3. Run standard LLVM IR optimizations
2.4. Codegen LLVM IR to PTX with NVPTX backend

14-stencil test suite: benchmark

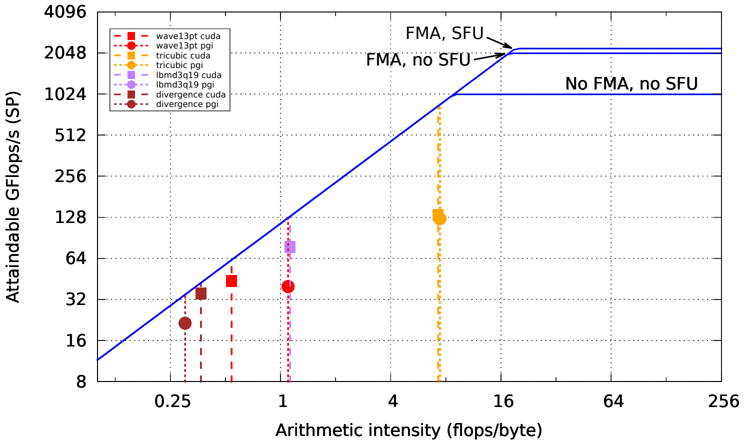
KernelGen vs PGI OpenACC vs CAPS OpenACC on GTX 680



- Tests precision mode: **single**
- Software: KernelGen r1780, PGI OpenACC 13.2, CAPS 3.2.4
- Hardware: NVIDIA GTX 680 (GK104, sm_30)
- Speed-up values: above 1 – KernelGen's kernel is faster than OpenACC's, below 1 – OpenACC's kernel is faster than KernelGen's (on the same GPU)
- Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

14-stencil test suite: roofline

The test suite contains stencils with different arithmetic intensities:



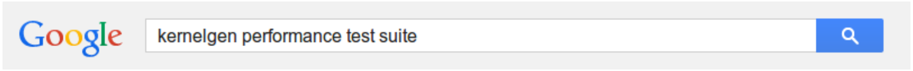
- Hardware: NVIDIA GTX 680M (GK104, sm_30)

Conclusions

1. The most frequently cited shared memory optimizations are not beneficial on modern cache-enabled GPUs
2. Vectorization of loads/stores gives 10-15% improvement, and does not seem to be well-known (the only prior reference is a recent brief blog entry)
3. Compared to OpenACC commercial compilers, polyhedral toolchains can provide competitive performance for stencil codes
4. Open design toolchains could generate even better code with the new optimizations we've investigated
5. Yet to be addressed: how to efficiently transform for GPU codes with non-coalesced accesses in stencils

Collaboration proposal

- Joint implementation of vectorization and aligning in open-source polyhedral compiler
- Joint work on tests, targets and optimizations research for a CUDA/OpenACC stencil benchmark:


kernelgen performance test suite

Thanks!

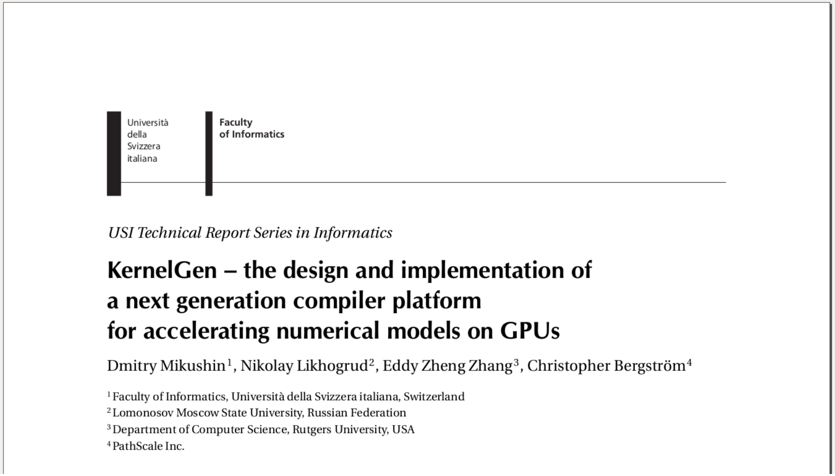
This presentation is supported by

EXA2CT: Exascale Algorithms and Advanced Computational Techniques (EU FP7-ICT programme)

Hardware access is provided by

CSCS: Swiss National Supercomputing Centre

tesla-cmc: GPU server @ Lomonosov Moscow State University



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.