KernelGen – A prototype of auto-parallelizing Fortran/C compiler for NVIDIA GPUs

Dmitry Mikushin[1,3] Nikolay Likhogrud[2,3] Hou Yunqing[4] Sergey Kovylov[5]

[1]Institute of Computational Science, University of Lugano

[2]Lomonosov Moscow State University

[3]Applied Parallel Computing LLC

[4]Nanyang Technological University

[5]NVIDIA

KernelGen research project

Goals:

- Conserve the original application source code, keep all GPU-specific things in the background
- Minimize manual work on specific code $\Rightarrow$ develop a compiler toolchain usable with many models

KernelGen research project

Goals:

- Conserve the original application source code, keep all GPU-specific things in the background
- Minimize manual work on specific code $\Rightarrow$ develop a compiler toolchain usable with many models

Rationale:

- Old good programming languages could still be usable, if accurate code analysis & parallelization methods exist
- OpenACC is too restrictive for complex apps and needs more flexibility
- GPU tends to become a central processing unit in near future, contradicting with OpenACC paradigm
- NWP is a perfect testbed for novel accelerator programming models

WRF specifics

- Sets of multiple numerical blocks to switch between, depending on model purpose $\Rightarrow$ no need to compile all code for GPU at time, JIT-compile only used parts
- Complex compilation system, most of code is compiled to static libraries, many potential GPU kernels have external dependencies $\Rightarrow$ needs modified linker to resolve kernels dependencies at link time

Project Team

Università della Svizzera italiana — Facoltà di scienze informatiche

University of Lugano,
Institute of Computational Science

Lomonosov Moscow State University,
Faculty of Computational Mathematics and Cybernetics

Applied Parallel Computing LLC

Project Team

Università della Svizzera italiana

Facoltà di scienze informatiche

University of Lugano,
Institute of Computational Science

Lomonosov Moscow State University,
Faculty of Computational Mathematics and Cybernetics

Applied Parallel Computing LLC

With technical support of many communities:

GCC

Polly

Polyhedral LLVM

NVIDIA

+ AsFermi, OpenMPI and others

Project state in September 2011 (v0.1)

Results:

- Could successfully generate CUDA and OpenCL kernels out of parallel loops in Fortran, with lots of limitations
- Automatic handling of host-device data transfers, with all process data kept on host
- Better language support than F2C-ACC, but still a lot of issues

Project state in September 2011 (v0.1)

Results:

- Could successfully generate CUDA and OpenCL kernels out of parallel loops in Fortran, with lots of limitations
- Automatic handling of host-device data transfers, with all process data kept on host
- Better language support than F2C-ACC, but still a lot of issues

Implementation:

- Pretty-printed AST – to markup and transform code into host and device parts
- No reliable data dependency analysis in loops
- LLVM + C Backend – to convert Fortran to C and chain to CUDA compiler

Project state in September **2012** (v0.2_nvptx)

Results:

- Can analyze arbitrary loops in C/C++/Fortran for parallelism and generate CUDA kernels
- Better quality of parallelism detection, than OpenACC from PGI
- Automatic handling of host-device data transfers, with all process data kept on device
- Full compatibility with conventional GCC compiler and linker

Project state in September **2012** (v0.2_nvptx)

Results:

- Can analyze arbitrary loops in C/C++/Fortran for parallelism and generate CUDA kernels
- Better quality of parallelism detection, than OpenACC from PGI
- Automatic handling of host-device data transfers, with all process data kept on device
- Full compatibility with conventional GCC compiler and linker

Implementation:

- DragonEgg – to emit LLVM IR from C/C++/Fortran
- LLVM loop extractor pass – to detect loops in compile time
- Modified LLVM Polly – to perform loop analysis in runtime
- LLVM NVPTX Backend – to emit PTX ISA directly from LLVM IR
- Modified GCC compiler and custom LTO wrapper – to support calling external functions in loops and link code from static libraries

KernelGen user interface design

- KernelGen is based on GCC and is fully compatible with it
- Executable binary preserves host-only version, that is used by default; GPU version is activated by request
- Execution mode is controlled by $kernelgen_runmode: 0 – run original CPU binary, 1 – run GPU version

```
$ NETCDF=/opt/kernelgen ./configure
Please select from among the following supported platforms.
...
  27.  Linux x86_64, kernelgen-gfortran compiler for CUDA   (serial)
  28.  Linux x86_64, kernelgen-gfortran compiler for CUDA   (smpar)
  29.  Linux x86_64, kernelgen-gfortran compiler for CUDA   (dmpar)
  30.  Linux x86_64, kernelgen-gfortran compiler for CUDA   (dm+sm)
Enter selection [1-38] : 27
...
$ ./compile em_real
...
$ cd test/em_real/
$ kernelgen_runmode=1 ./real.exe
```

OpenACC: no external calls

OpenACC compilers do not allow calls from different compilation units:

sincos.f90

```
!$acc parallel
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sincos_ijk(x(i, j, k), y(i, j, k))
    enddo
  enddo
enddo
!$acc end parallel
```

function.f90

```
sincos_ijk = sin(x) + cos(y)
```

```
pgfortran -fast -Mnomain -Minfo=accel -ta=nvidia,time -Mcuda=keepgpu,keepbin,keepptx,ptxinfo -c ../sincos.f90 -o ↩
      sincos.o
PGF90-W-0155-Accelerator region ignored; see -Minfo messages  (../sincos.f90: 33)
sincos:
     33, Accelerator region ignored
     36, Accelerator restriction: function/procedure calls are not supported
     37, Accelerator restriction: unsupported call to sincos_ijk
  0 inform,   1 warnings,   0 severes, 0 fatal for sincos
```

KernelGen: external calls

Dependency resolution during linking
Kernels generation in runtime
} ⇒ Support for external calls defined in other objects or static libraries

```
!$acc parallel
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sincos_ijk(x(i, j, k), y(i, j, k))
    enddo
  enddo
enddo
!$acc end parallel
```

```
sincos_ijk = sin(x) + cos(y)
```

result

```
Launching kernel __kernelgen_sincos__loop_3
    blockDim = { 32, 16, 1 }
    gridDim = { 16, 32, 63 }
Finishing kernel __kernelgen_sincos__loop_3
__kernelgen_sincos__loop_3 time = 0.00536099 sec
```

OpenACC: no pointers tracking

In Fortran allocatable arrays carry their dimensions. Not the case in C:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy)
{
    #pragma acc parallel
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
            {
                int idx = i + nx * j + nx * ny * k;
                xy[idx] = sin(x[idx]) + cos(y[idx]);
            }
    ...
}
```

```
pgcc -fast -Minfo=accel -ta=nvidia,time -Mcuda=keepgpu,keepbin,keepptx,ptxinfo -c ../sincos.c -o sincos.o
PGC-W-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-↩
       variable index for symbol (../sincos.c: 27)
sincos:
     27, Accelerator kernel generated
     28, Complex loop carried dependence of *(y) prevents parallelization
         Complex loop carried dependence of *(x) prevents parallelization
         Complex loop carried dependence of *(xy) prevents parallelization
...
     30, Accelerator restriction: size of the GPU copy of xy is unknown
...
```

KernelGen: smart pointers tracking

Pointer alias analysis is performed in runtime, assisted with addresses substitution.

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy)
{
    #pragma acc parallel
    for (int k = 0; k < nz; k++)
        for (int j = 0; j < ny; j++)
            for (int i = 0; i < nx; i++)
            {
                int idx = i + nx * j + nx * ny * k;
                xy[idx] = sin(x[idx]) + cos(y[idx]);
            }
    ...
}
```

result

```
Launching kernel __kernelgen_sincos_loop_8.preheader
    blockDim = { 32, 16, 1 }
    gridDim = { 16, 32, 63 }
Finishing kernel __kernelgen_sincos_loop_8.preheader
__kernelgen_sincos_loop_8.preheader time = 0.00528601 sec
```

KernelGen: can parallelize while loops

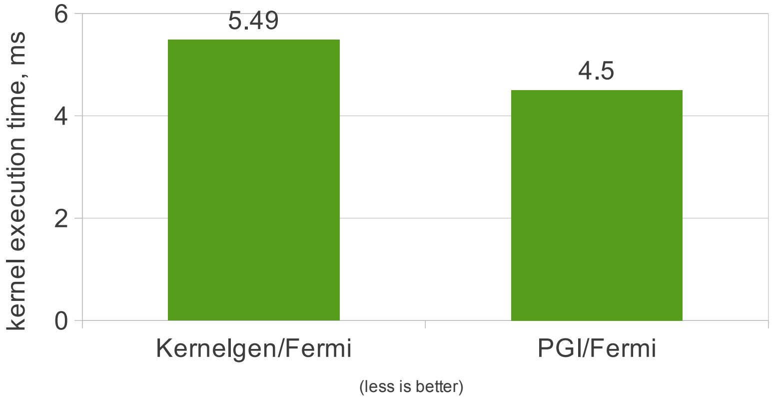
Thanks to the nature of LLVM and Polly, KernelGen can parallelize while-loops semantically equivalent to for-s (OpenACC can't):

```
i = 1
do while (i .le. nx)
  j = 1
  do while (j .le. nz)
    k = 1
    do while (k .le. ny)
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
      k = k + 1
    enddo
    j = j + 1
  enddo
  i = i + 1
enddo
```

```
Launching kernel __kernelgen_matmul__loop_9
    blockDim = { 32, 32, 1 }
    gridDim = { 2, 16, 1 }
Finishing kernel __kernelgen_matmul__loop_9
__kernelgen_matmul__loop_9 time = 0.00953514 sec
```

Benchmarking: sincos

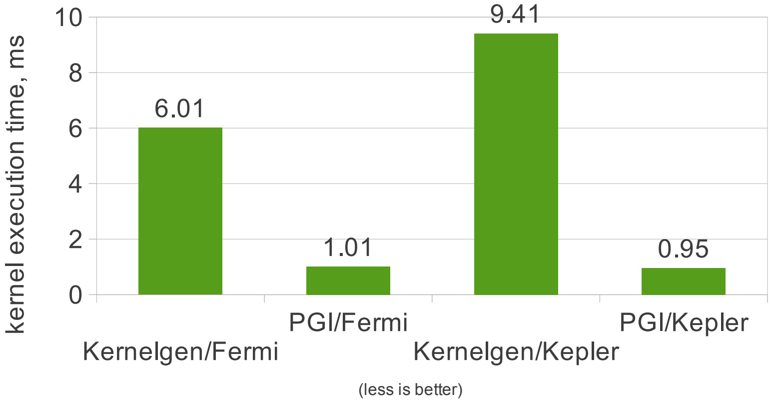
xy[i,j,k] := sin(x[i,j,k]) + cos(y[i,j,k])



PGI 12.6, Fermi – Tesla C2050

Benchmarking: matmul

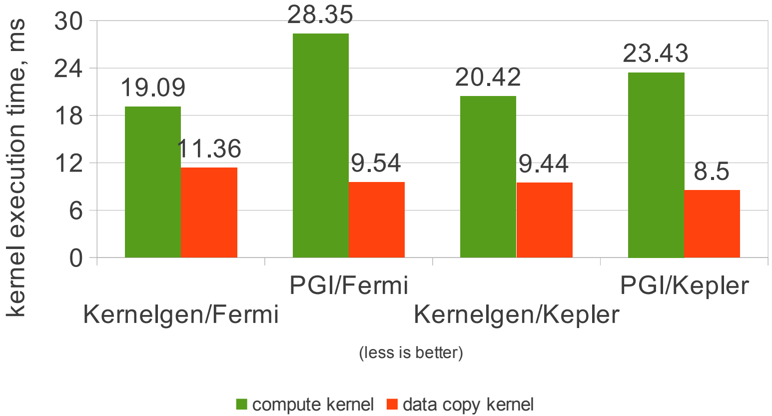
PGI is currently faster because of partial reduction in registers:



PGI 12.6, Fermi – Tesla C2050, Kepler – GTX 680M

Benchmarking: jacobi

On finite-difference patterns KernelGen performance is better:



PGI 12.6, Fermi – Tesla C2050, Kepler – GTX 680M

KernelGen concepts

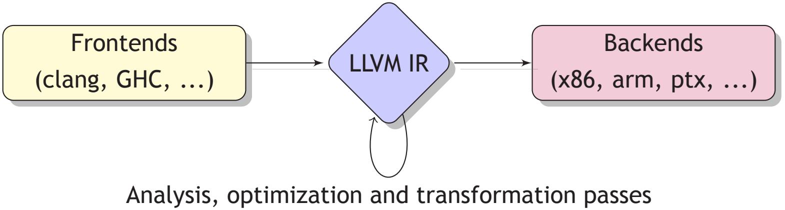
- Main GPU and peripheral host-system: initially port on GPU as much parallel code as possible, without human decision
- Fallback to CPU version in case of calls to host-only functions (I/O, syscalls, ...) or non-parallel loops or inefficient parallel code
- Perform transparent host-device data sharing on-demand, keeping all data on device by default, rather than on host

KernelGen concepts

- Main GPU and peripheral host-system: initially port on GPU as much parallel code as possible, without human decision
- Fallback to CPU version in case of calls to host-only functions (I/O, syscalls, ...) or non-parallel loops or inefficient parallel code
- Perform transparent host-device data sharing on-demand, keeping all data on device by default, rather than on host
- Use GCC frontends to support major programming languages (Fortran, C/C++, Ada, etc.)
- Unify all languages to the common intermediate representation
- Extract potentially parallel loops into kernels during compile-time, but decide the execution mode, taking in account runtime information (JIT)
- Adjust kernel execution mode, using the dynamically collected statistics or use profile files from previous runs

LLVM for Fortran & GPU in a nutshell

- LLVM – a universal system of programs analysis, transformation and optimization with RISC-like intermediate representation (LLVM IR SSA)



LLVM for Fortran & GPU in a nutshell

Consider the following kernel written in Fortran:

```fortran
subroutine sum_kernel(a, b, c, length)
implicit none

integer :: length
real, dimension(length) :: a, b, c
integer :: idx, threadIdx_x

  idx = threadIdx_x() + 1

  c(idx) = a(idx) + b(idx)

end subroutine sum_kernel
```

LLVM for Fortran & GPU in a nutshell

With help of GCC and DragonEgg it could be translated into LLVM IR:

```
$ kernelgen-dragonegg kernel.f90 -o - | opt -O3 -S -o kernel.ll
```

```
target datalayout = "e-p:64:64:64-S128-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f16:16:16-f32:32:32-f64:64:64-↩
      f128:128:128-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64"
target triple = "x86_64-unknown-linux-gnu"

define void @sum_kernel_([0 x float]* noalias nocapture %a, [0 x float]* noalias nocapture %b, [0 x float]* ↩
       noalias nocapture %c, i32* noalias nocapture %length) nounwind uwtable {
entry:
  %0 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x() nounwind
  %1 = add i32 %0, 1
  %2 = sext i32 %1 to i64
  %3 = add i64 %2, -1
  %4 = getelementptr [0 x float]* %a, i64 0, i64 %3
  %5 = load float* %4, align 4
  %6 = getelementptr [0 x float]* %b, i64 0, i64 %3
  %7 = load float* %6, align 4
  %8 = fadd float %5, %7
  %9 = getelementptr [0 x float]* %c, i64 0, i64 %3
  store float %8, float* %9, align 4
  ret void
}

declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()
```

LLVM for Fortran & GPU in a nutshell

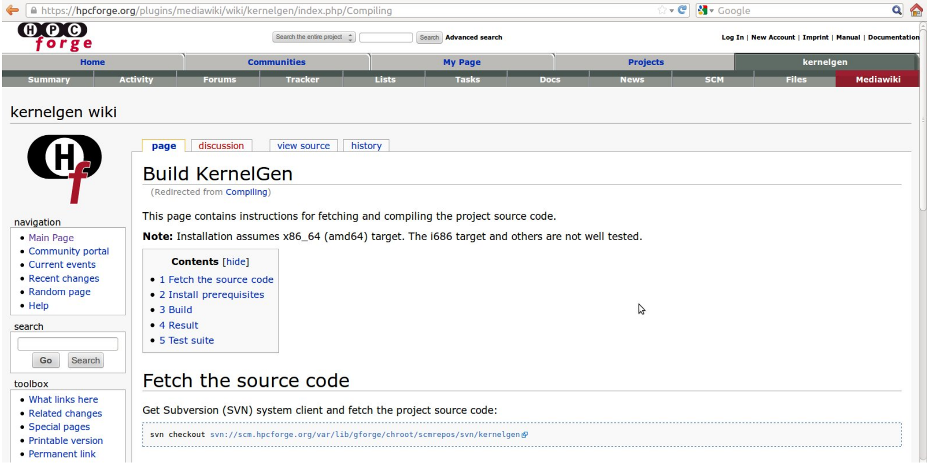
PTX GPU assembly can be emitted from LLVM IR with help of NVPTX backend:

```
$ llc -march="nvptx64" -mcpu="sm_30" kernel.ll -o kernel.ptx
```

```
.func sum_kernel_(.param .b64 sum_kernel__param_0, .param .b64 sum_kernel__param_1, .param .b64 ↩
      sum_kernel__param_2, .param .b64 sum_kernel__param_3) {
.reg .pred %p<396>;
.reg .s16 %rc<396>;
.reg .s16 %rs<396>;
.reg .s32 %r<396>;
.reg .s64 %rl<396>;
.reg .f32 %f<396>;
.reg .f64 %fl<396>;
mov.u32  %r0, %tid.x;
add.s32  %r0, %r0, 1;
cvt.s64.s32  %rl0, %r0;
add.s64  %rl0, %rl0, -1;
shl.b64  %rl0, %rl0, 2;
ld.param.u64  %rl1, [sum_kernel__param_0];
add.s64  %rl1, %rl1, %rl0;
ld.param.u64  %rl2, [sum_kernel__param_1];
add.s64  %rl2, %rl2, %rl0;
ld.f32  %f0, [%rl2];
ld.f32  %f1, [%rl1];
add.f32  %f0, %f1, %f0;
ld.param.u64  %rl1, [sum_kernel__param_2];
add.s64  %rl0, %rl1, %rl0;
st.f32  [%rl0], %f0;
ret;
}
```

http://kernelgen.org/testit/

- Please help us to improve the quality and usefulness of KernelGen
- The code is open-source and could be easily compiled into binary package



Technical plan for Stage 3 (Fall 2012)

Compiler core improvements (by priority):

1. Get rid of code inlining before applying loops analysis with Polly
2. Fix crashes of kernels using CUDA math functions on Kepler
3. Solve problems with compilation of big kernels using ptxas
4. Rewrite gpu-cpu data sharing model more efficiently
5. Replace host-assisted loop kernels launching with Kepler K20's dynamic parallelism
6. Enable Polly tiling with support of shared memory, loops interchanging and Kepler's warp shuffle

Improve usability:

- Create Ubuntu PPA repository shipping KernelGen compiler binaries
- Testing: NPB, polybench, COSMO radiation, WRF

Download link for this presentation:
http://kernelgen.org/ncar2012/

Project mailing list:
kernelgen-devel@lists.hpcforge.org

Thank you! ☺