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

Dmitry Mikushin Nikolay Likhogrud Sergey Kovylov

KernelGen compiler 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 existing computational applications

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

Simple example: Fortran

```fortran
program demo

integer :: argc, nx, ny, ns
character(len=32) :: arg
real, allocatable, dimension(:,:,:) :: x, y, xy
real :: start, finish

! Read arguments
call get_command_argument(1, arg)
read(arg, '(I32)') nx
call get_command_argument(2, arg)
read(arg, '(I32)') ny
call get_command_argument(3, arg)
read(arg, '(I32)') ns

! Allocate data arrays.
allocate(x(nx,ny,ns), y(nx,ny,ns), xy(nx,ny,ns))

! Initialize arrays.
x = atan(1.0)
y = x

! Computational loop
call cpu_time(start)
do k = 1, ns
  do j = 1, ny
    do i = 1, nx
      xy(i,j,k) = asin(sin(x(i,j,k))) + acos(cos(y(i,j,k)))
    enddo
  enddo
enddo
call cpu_time(finish)

write(*,*) 'compute time = ', finish - start

write(*,*) 'maxval = ', maxval(xy), &
  'minval = ', minval(xy)

! Deallocate arrays.
deallocate(x, y, xy)

end program demo
```

Simple example: Fortran

Compile the Fortran example as usual, just use *kernelgen-gfortran* instead of *gfortran*:

```
$ kernelgen-gfortran -O3 example_f.f90 -o example_f
```

KernelGen always generates binaries usable both on CPU and GPU. To execute the regular version, run it as usual:

```
$ ./example_f 512 256 256
 compute time =     1.8481150
 maxval =     1.5707963     minval =     1.5707963
```

In order to run GPU-accelerated version, simply set *kernelgen_runmode* environment variable to 1:

```
$ kernelgen_runmode=1 ./example_f 512 256 256
 compute time =     0.28801799
 maxval =     1.5707964     minval =     1.5707964
```

Simple example: C

```c
#include <malloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(int argc, char* argv[]) {

int nx = atoi(argv[1]);
int ny = atoi(argv[2]);
int ns = atoi(argv[3]);

size_t szarray = nx * ny * ns;
size_t szarrayb = szarray * sizeof(float);

float* x = (float*)malloc(szarrayb);
float* y = (float*)malloc(szarrayb);
float* xy = (float*)malloc(szarrayb);

for (int i = 0; i < szarray; i++)
{
  x[i] = atan(1.0);
  y[i] = x[i];
}
struct timeval start, finish;
```

```c
gettimeofday(&start, NULL);
for (int k = 0; k < ns; k++)
  for (int j = 0; j < ny; j++)
    for (int i = 0; i < nx; i++)
    {
      int idx = i + nx * (j + ny * k);
      xy[idx] = asinf(sinf(x[idx])) + acosf(cosf(y[idx]));
    }
gettimeofday(&finish, NULL);
printf("compute time = %f\n",
  get_time_diff(&start, &finish));

float minval = xy[0], maxval = xy[0];
for (int i = 0; i < szarray; i++)
{
  if (minval > xy[i]) minval = xy[i];
  if (maxval < xy[i]) maxval = xy[i];
}
printf("maxval = %f, minval = %f\n", maxval, minval);

// Deallocate arrays.
free(x); free(y); free(xy);

return 0;
}
```

Simple example: C

Compile the Fortran example as usual, just use *kernelgen-gcc* instead of *gcc*:

```
$ kernelgen-gcc -O3 -std=c99 example_c.c -o example_c
```

KernelGen always generates binaries usable both on CPU and GPU. To execute the regular version, run it as usual:

```
$ ./example_c 512 256 256
compute time = 1.848575
maxval = 1.570796, minval = 1.570796
```

In order to run GPU-accelerated version, simply set *kernelgen_runmode* environment variable to 1:

```
$ kernelgen_runmode=1 ./example_c 512 256 256
compute time = 0.293359
maxval = 1.570796, minval = 1.570796
```

Current limitations of OpenACC compilers

OpenACC currently focuses on immediate acceleration benefits in small and average codes, and could hardly be shifted to broader scope without addressing the following limitations:

- **External calls** – support of external calls in accelerated loops is a must for big projects, where functionality is distributed between multiple code units, for instance, Fortran modules.
- **Pointers analysis** – in many circumstances compiler cannot reliably determine the relationship between pointers and memory access ranges, requiring user to provide additional information with directives.

$\Rightarrow$ Should one sacrifice code design and integrity for the benefit of acceleration?

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

```
function sincos_ijk(x, y)

  implicit none
  real, intent(in) :: x, y
  real :: sincos_ijk

  sincos_ijk = sin(x) + cos(y)

end function sincos_ijk
```

```
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
```

KernelGen: can handle external calls

Dependency resolution during linking
Kernels generation in runtime
$\Rightarrow$
Support for external calls defined in other objects or static libraries

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

```
function sincos_ijk(x, y)

  implicit none
  real, intent(in) :: x, y
  real :: sincos_ijk

  sincos_ijk = sin(x) + cos(y)

end function sincos_ijk
```

```
Launching kernel __kernelgen_sincos__loop_3
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos__loop_3
__kernelgen_sincos__loop_3 time = 4.986428e-03 sec
```

KernelGen: and external static libraries

Thanks to cutomized LTO wrapper, KernelGen can extract kernels dependencies from static libraries:

```
kernelgen-gcc -std=c99 -c ../main.c -o main.o
kernelgen-gfortran -c ../sincos.f90 -o sincos.o
kernelgen-gfortran -c ../function.f90 -o function.o
ar rcs libfunction.a function.o
kernelgen-gfortran main.o sincos.o -o function -L. -lfunction
```

```
$ kernelgen_runmode=1 ./function 512 512 64
__kernelgen_sincos__loop_3: regcount = 22, size = 512
Loaded '__kernelgen_sincos__loop_3' at: 0xc178f0
Launching kernel __kernelgen_sincos__loop_3
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos__loop_3
__kernelgen_sincos__loop_3 time = 4.974710e-03 sec
```

OpenACC: no pointers tracking

Compiler cannot determine relationships between pointers and data ranges:

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:

```
for (c2=0;c2<=63;c2++) {
  for (c4=0;c4<=511;c4++) {
    for (c6=0;c6<=511;c6++) {
      Stmt__5_cloned_(c2,c4,c6);
    }
  }
}
```

```
Statements {
     Stmt__5_cloned_
            Domain      := { Stmt__5_cloned_[i0, i1, i2] : i0 >= 0 and i0 <= 63 and i1 >= 0 and i1 <= 511 and i2 >= 0 and ↩
                  i2 <= 511 };
            Scattering  := { Stmt__5_cloned_[i0, i1, i2] -> scattering[0, i0, 0, i1, 0, i2, 0] };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47246749696 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47246749699 + 1048576i0 + 2048i1 + 4i2 };
            ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47313862656 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47313862659 + 1048576i0 + 2048i1 + 4i2 };
            WriteAccess := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47380975616 + 1048576i0 + 2048i1 + 4i2 and o0↩
                  <= 47380975619 + 1048576i0 + 2048i1 + 4i2 };
}
```

KernelGen: smart pointers tracking

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

```
for (c2=0;c2<=63;c2++) {
  for (c4=0;c4<=511;c4++) {
    for (c6=0;c6<=511;c6++) {
      Stmt__5_cloned_(c2,c4,c6);
    }
  }
}
```

Static Control Part (SCoP) representation of the loop kernel

```
Statements {
     Stmt__5_cloned_
             Domain      := { Stmt__5_cloned_[i0, i1, i2] : i0 >= 0 and i0 <= 63 and i1 >= 0 and i1 <= 511 and i2 >= 0 and ↩
                  i2 <= 511 };
             Scattering  := { Stmt__5_cloned_[i0, i1, i2] -> scattering[0, i0, 0, i1, 0, i2, 0] };
             ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47246749696 + 1048576i0 + 2048i1 + 4i2 and o0↩
                    <= 47246749699 + 1048576i0 + 2048i1 + 4i2 };
             ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47313862656 + 1048576i0 + 2048i1 + 4i2 and o0↩
                    <= 47313862659 + 1048576i0 + 2048i1 + 4i2 };
             WriteAccess := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47380975616 + 1048576i0 + 2048i1 + 4i2 and o0↩
                    <= 47380975619 + 1048576i0 + 2048i1 + 4i2 };
}
```

KernelGen: smart pointers tracking

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

```
for (c2=0;c2<=63;c2++) {
  for (c4=0;c4<=511;c4++) {
    for (c6=0;c6<=511;c6++) {
      Stmt__5_cloned_(c2,c4,c6);
    }
  }
}
```

Static Control Part (SCoP) representation of the loop kernel

```
Statements {
     Stmt__5_cloned_
               Domain      := { Stmt__5_cloned_[i0, i1, i2] : i0 >= 0 and i0 <= 63 and i1 >= 0 and i1 <= 511 and i2 >= 0 and ↩
                    i2 <= 511 };
               Scattering  := { Stmt__5_cloned_[i0, i1, i2] -> scattering[0, i0, 0, i1, 0, i2, 0] };
               ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47246749696 + 1048576i0 + 2048i1 + 4i2 and o0↩
                      <= 47246749699 + 1048576i0 + 2048i1 + 4i2 };
               ReadAccess  := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47313862656 + 1048576i0 + 2048i1 + 4i2 and o0↩
                      <= 47313862659 + 1048576i0 + 2048i1 + 4i2 };
               WriteAccess := { Stmt__5_cloned_[i0, i1, i2] -> NULL[o0] : o0 >= 47380975616 + 1048576i0 + 2048i1 + 4i2 and o0↩
                      <= 47380975619 + 1048576i0 + 2048i1 + 4i2 };
}
```

Data read/write access analysis after substituting pointers and constants

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_10
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos_loop_10
__kernelgen_sincos_loop_10 time = 2.300006e-02 sec
```

Other nice features

Comfortable acceleration framework should be designed, taking in account properties of the supported languages and common habits of developers, for instance:

- Handle different forms of loops, not only arithmetic for/do
- Parallelize implicit loops (Fortran array-wise statements, elemental functions, etc.)
- Pointer arithmetics (C/C++)
- ...

Parallelizing while-loops

Thanks to the nature of LLVM and Polly, KernelGen can parallelize while-loops *semantically equivalent* to for-loops (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
```

Parallelizing Fortran array-wise code

Fortran array-wise statements are expanded into plain loops inside compiler frontend.
For this reason, KernelGen is able to parallelize them into GPU kernels, for instance:

```
1  program demo
2
3  implicit none
4  integer :: n
5  complex*16, allocatable, dimension(:) :: c1, c2,↩
       z
6  character(len=128) :: arg
7  integer :: i
8  real*8 :: v1, v2
9  real :: start, finish
10
11 call get_command_argument(1, arg)
12 read(arg, '(I64)') n
13
14 ! Allocate data arrays.
15 allocate(c1(n), c2(n), z(n))
16
17 ! Initialize arrays.
18 do i = 1, n
19   call random_number(v1)
20   call random_number(v2)
21   c1(i) = cmplx(v1, v2)
22   call random_number(v1)
23   call random_number(v2)
24   c2(i) = cmplx(v1, v2)
25 enddo
26
27 ! Implicit computational loop
28 call cpu_time(start)
29 z = conjg(c1) * c2
30 call cpu_time(finish)
31
32 write(*,*) 'compute time = ', finish - start
33
34 print *, 'z min = (', minval(realpart(z)), &
35   ', ', minval(imagpart(z)), '), max = (', &
36   maxval(realpart(z)), ', ', minval(imagpart(z)), ')'
37
38 ! Deallocate arrays.
39 deallocate(c1, c2, z)
40
41 end program demo
```

Parallelizing Fortran array-wise code

Compile the Fortran example as usual, just use *kernelgen-gfortran* instead of *gfortran*:

```
$ kernelgen-gfortran -O3 -std=c99 conjg.f90 -o conjg
```

```
$ ./conjg $((256*256*256))
 compute time =   0.10800600
 z min = (  7.18319034686704879E-006 ,  -0.99788337878880551      ), max = (  ↩
      1.9723582564715194      ,  -0.99788337878880551      )
```

```
$ kernelgen_runmode=1 ./conjg $((256*256*256))
 compute time =   2.80020237E-02
 z min = (  7.18319034686704879E-006 ,  -0.99788337878880551      ), max = (  ↩
      1.9723582564715194      ,  -0.99788337878880551      )
```

OpenACC: no pointer arithmetics

Compiler cannot parallelize loops containing pointer arithmetics:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    float *xp = x, *yp = y, *xyp = 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++)
                *(xyp++) = sin(*(xp++)) + cos(*(yp++));
}
```

```
$ make
pgcc -fast -Minfo=accel -ta=nvidia,time -Mcuda=keepgpu,keepbin,keepptx,ptxinfo -c ../sincos.c -o sincos.o
PGC-W-0155-Pointer assignments are not supported in accelerator regions:  xyp (../sincos.c: 34)
PGC-W-0155-Accelerator region ignored  (../sincos.c: 29)
PGC/x86-64 Linux 13.2-0: compilation completed with warnings
```

KernelGen: pointer arithmetics support

In LLVM/Polly used by KernelGen all arrays accesses are lowered to pointers, thus there is no difference between indexed arrays and pointer arithmetics, both are supported:

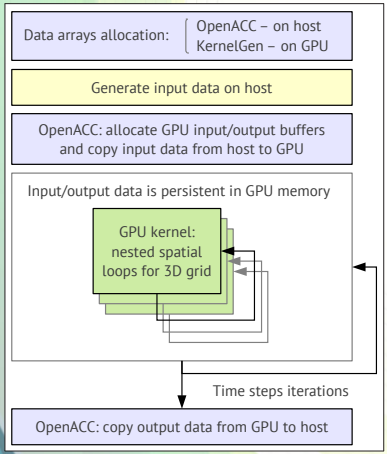
sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    float *xp = x, *yp = y, *xyp = 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++)
                *(xyp++) = sin(*(xp++)) + cos(*(yp++));
}
```

```
Launching kernel __kernelgen_sincos_loop_10
    blockDim = { 32, 4, 4 }
    gridDim = { 16, 128, 16 }
Finishing kernel __kernelgen_sincos_loop_10
__kernelgen_sincos_loop_10 time = 2.298868e-02 sec
```

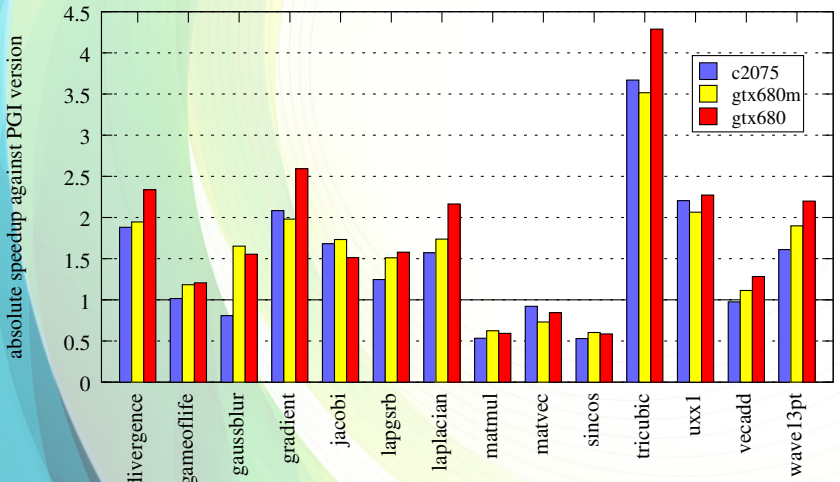
Performance test suite



The test suite is composed out of similarly organized small applications, simulating typical numerical model behavior:

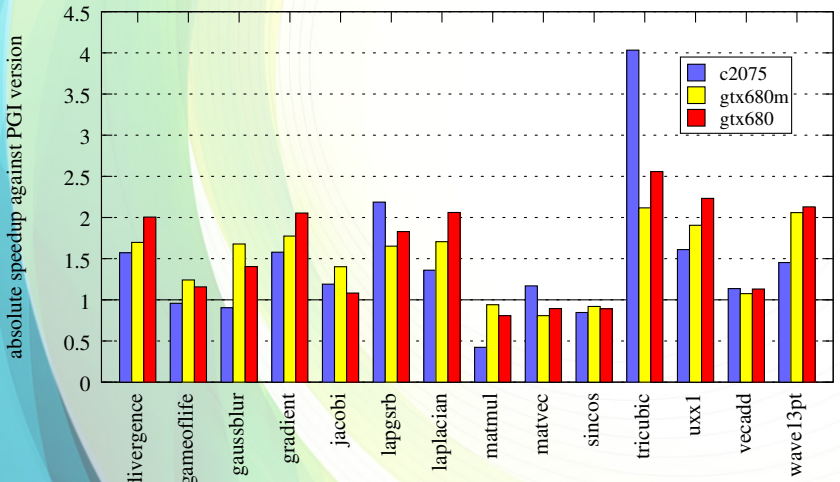
- Outer time iterations loop (sequential)
- Inner 2D or 3D spatial loops on regular grid (parallel)
- Input/output data is persistent in GPU memory during time iterations
- Results correctness in checked by computing means/norms of output data

Performance: KernelGen vs PGI OpenACC



- Tests precision mode: **single**
- Software: KernelGen r1740, PGI OpenACC 13.2
- Hardware: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30)
- Values above 1 – KernelGen kernel is faster than PGI, values below 1 – PGI kernel is faster than KernelGen (on the same GPU)
- Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

Performance: KernelGen vs PGI OpenACC



- Tests precision mode: **double**
- Software: KernelGen r1740, PGI OpenACC 13.2
- Hardware: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30)
- Values above 1 – KernelGen kernel is faster than PGI, values below 1 – PGI kernel is faster than KernelGen (on the same GPU)
- Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test

Performance factors: cache config

1. **Cache config** Since current kernels generator does not utilize shared memory, in KernelGen cache mode is configured to larger L1 cache, giving some additional speedup

```
// Since KernelGen does not utilize shared memory at the moment,
// use larger L1 cache by default.
CU_SAFE_CALL(cuCtxSetCacheConfig(CU_FUNC_CACHE_PREFER_L1));
```

Performance factors: compute grid

2 KernelGen uses $\{128, 1, 1\}$ blocks and 3D grid, while PGI uses $\{128, 1\}$ blocks and 2D grid. Previously we used $\{32, 4, 4\}$, which was slower.

PGI and KernelGen profilers reports for divergence test, single precision, $512 \times 256 \times 256$ problem:

```
Accelerator Kernel Timing data
/home/marcusmae/forge/kernelgen/tests/perf/divergence/pgi/../divergence.c
  divergence  NVIDIA  devicenum=0
        time(us): 154,660
        62: kernel launched 10 times
            grid: [4x254]  block: [128]
             device time(us): total=154,660 max=15,560 min=15,373 avg=15,466
            elapsed time(us): total=154,742 max=15,570 min=15,381 avg=15,474
```

```
Kernel function call __kernelgen_divergence_loop_10
__kernelgen_divergence_loop_10 @ 0xeb32b61a9530d97f53f77c5abbd67132
Launching kernel __kernelgen_divergence_loop_10
    blockDim = { 128, 1, 1 }
    gridDim = { 4, 254, 254 }
Finishing kernel __kernelgen_divergence_loop_10
__kernelgen_divergence_loop_10 time = 7.912811e-03 sec
```

Performance factors: code optimization

3 KernelGen under-optimizes GPU math (sincos) and stores in reduction (matmul, matvec):

```
CUDA.LoopHeader.x.preheader:                      ; preds = %"Loop Function Root"
  %p_newGEPInst.cloned = getelementptr float* inttoptr (i64 47380979712 to float*)
  store float 0.000000e+00, float * %p_newGEPInst.cloned
  %p_.moved.to.4.cloned = shl nsw i64 %3, 9
  br label %polly.loop_body
CUDA.AfterLoop.x:                                 ; preds = %polly.loop_body, %"Loop Function Root"
  ret void
polly.loop_body:                                  ; preds = %polly.loop_body, %CUDA.LoopHeader.x.preheader
  %_p_scalar_ = phi float [ 0.000000e+00, %CUDA.LoopHeader.x.preheader ], [ %p_8, %polly.loop_body ]
  %polly.loopiv10 = phi i64 [ 0, %CUDA.LoopHeader.x.preheader ], [ %polly.next_loopiv, %polly.loop_body ]
  %polly.next_loopiv = add i64 %polly.loopiv10, 1
  %p_ = add i64 %polly.loopiv10, %p_.moved.to.4.cloned
  %p_newGEPInst9.cloned = getelementptr float* inttoptr (i64 47246749696 to float*), i64 %p_
  %p_newGEPInst12.cloned = getelementptr float* inttoptr (i64 47380971520 to float*), i64 %polly.loopiv10
  %_p_scalar_5 = load float* %p_newGEPInst9.cloned
  %_p_scalar_6 = load float* %p_newGEPInst12.cloned
  %p_7 = fmul float %_p_scalar_5, %_p_scalar_6
  %p_8 = fadd float %_p_scalar_, %p_7
  store float %p_8, float* %p_newGEPInst.cloned
  %exitcond = icmp eq i64 %polly.next_loopiv, 512
  br i1 %exitcond, label %CUDA.AfterLoop.x, label %polly.loop_body
```

Performance factors: code optimization

3 KernelGen under-optimizes GPU math (sincos) and stores in reduction (matmul, matvec):

```
CUDA.LoopHeader.x.preheader:                      ; preds = %"Loop Function Root"
  %p_newGEPInst.cloned = getelementptr float* inttoptr (i64 47380979712 to float*)
  store float 0.000000e+00, float * %p_newGEPInst.cloned
  %p_.moved.to.4.cloned = shl nsw i64 %3, 9
  br label %polly.loop_body
CUDA.AfterLoop.x:                                 ; preds = %polly.loop_body, %"Loop Function Root"
  ret void
polly.loop_body:                                  ; preds = %polly.loop_body, %CUDA.LoopHeader.x.preheader
  %_p_scalar_ = phi float [ 0.000000e+00, %CUDA.LoopHeader.x.preheader ], [ %p_8, %polly.loop_body ]
  %polly.loopiv10 = phi i64 [ 0, %CUDA.LoopHeader.x.preheader ], [ %polly.next_loopiv, %polly.loop_body ]
  %polly.next_loopiv = add i64 %polly.loopiv10, 1
  %p_ = add i64 %polly.loopiv10, %p_.moved.to.4.cloned
  %p_newGEPInst9.cloned = getelementptr float* inttoptr (i
  %p_newGEPInst12.cloned = getelementptr float* inttoptr (
  %_p_scalar_5 = load float* %p_newGEPInst9.cloned
  %_p_scalar_6 = load float* %p_newGEPInst12.cloned
  %p_7 = fmul float %_p_scalar_5, %_p_scalar_6
  %p_8 = fadd float %_p_scalar_, %p_7
  store float %p_8, float* %p_newGEPInst.cloned
  %exitcond = icmp eq i64 %polly.next_loopiv, 512
  br i1 %exitcond, label %CUDA.AfterLoop.x, label %polly.loop_body
```

Marked out lines are header, tail and body of loop, as it looks like in LLVM IR

Performance factors: code optimization

3 KernelGen under-optimizes GPU math (sincos) and stores in reduction (matmul, matvec):

```
CUDA.LoopHeader.x.preheader:                      ; preds = %"Loop Function Root"
  %p_newGEPInst.cloned = getelementptr float* inttoptr (i64 47380979712 to float*)
  store float 0.000000e+00, float * %p_newGEPInst.cloned
  %p_.moved.to.4.cloned = shl nsw i64 %3, 9
  br label %polly.loop_body
CUDA.AfterLoop.x:                                 ; preds = %polly.loop_body, %"Loop Function Root"
  ret void
polly.loop_body:                                  ; preds = %polly.loop_body, %CUDA.LoopHeader.x.preheader
  %_p_scalar_ = phi float [ 0.000000e+00, %CUDA.LoopHeader
  %polly.loopiv10 = phi i64 [ 0, %CUDA.LoopHeader.x.prehea
  %polly.next_loopiv = add i64 %polly.loopiv10, 1
  %p_ = add i64 %polly.loopiv10, %p_.moved.to.4.cloned
  %p_newGEPInst9.cloned = getelementptr float* inttoptr (i
  %p_newGEPInst12.cloned = getelementptr float* inttoptr (
  %_p_scalar_5 = load float* %p_newGEPInst9.cloned
  %_p_scalar_6 = load float* %p_newGEPInst12.cloned
  %p_7 = fmul float %_p_scalar_5, %_p_scalar_6
  %p_8 = fadd float %_p_scalar_, %p_7
  store float %p_8, float* %p_newGEPInst.cloned
  %exitcond = icmp eq i64 %polly.next_loopiv, 512
  br i1 %exitcond, label %CUDA.AfterLoop.x, label %polly.loop_body
```

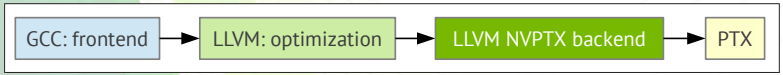
Properly optimized reduction should accumulate sum on register and store it (memory operation) at the end only once. Here store is performed on every iteration!

Performance factors: code generation

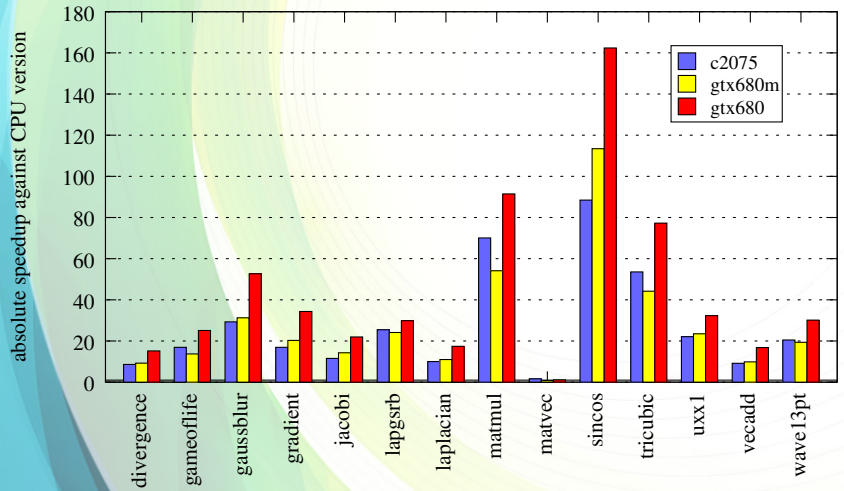
4 PGI is a source-to-source compiler, while KernelGen is a full compiler, thanks to LLVM NVPTX backend

PGI:

KernelGen:

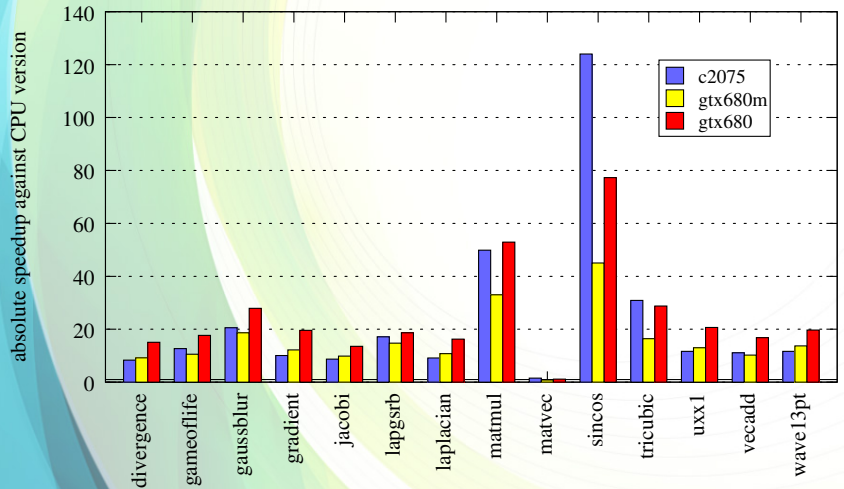


Performance: KernelGen vs CPU



- Tests precision mode: **single**
- Software: KernelGen r1740, GCC 4.6.4
- Hardware: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30), Intel Core i7-3610QM CPU 2.30GHz
- Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test
- Values above 1 – KernelGen GPU kernel is faster than GCC CPU kernel, values below 1 – GCC is faster than KernelGen
- CPU version is single-core

Performance: KernelGen vs CPU



- Tests precision mode: **double**
- Software: KernelGen r1740, GCC 4.6.4
- Hardware: NVIDIA Tesla C2075 (GF110, sm_20), NVIDIA GTX 680M (GK104, sm_30), NVIDIA GTX 680 (GK104, sm_30), Intel Core i7-3610QM CPU 2.30GHz
- Measurements are averaged from 10 invocations of all tests and 10 iterations inside every test
- Values above 1 – KernelGen GPU kernel is faster than GCC CPU kernel, values below 1 – GCC is faster than KernelGen
- CPU version is single-core

Parallelism detection

KernelGen analyses loops dependencies and maps parallel loops on GPU compute grid.
Sequential loops are kept unchanged, either inner or outer.

```
1  subroutine match_filter(HH, szhh, XX, szxx, YY, szyy)
2
3    implicit none
4    integer(kind=IKIND), intent(in) :: szhh, szxx, szyy
5    real(kind=RKIND), intent(in) :: HH(szhh), XX(szxx)
6    real(kind=RKIND), intent(out) :: YY(szyy)
7    integer(kind=IKIND) :: i, j
8    integer, parameter :: rkind = RKIND
9
10   ! This loop will be parallelized
11   do i = 1, szyy
12     YY(i) = 0.0_rkind
13     ! This loop will not be parallelized
14     do j = 1, szhh
15       YY(i) = YY(i) + XX(i + j - 1) * HH(j)
16     enddo
17   enddo
18
19 end subroutine match_filter
```

Parallelism detection

KernelGen analyses loops dependencies and maps parallel loops on GPU compute grid.
Sequential loops are kept unchanged, either inner or outer.

```
1  subroutine match_filter(HH, szhh, XX, szxx, YY, szyy)
2
3    implicit none
4    integer(kind=IKIND), intent(in) :: szhh, szxx, szyy
5    real(kind=RKIND), intent(in) :: HH(szhh), XX(szxx)
6    real(kind=RKIND), intent(out) :: YY(szyy)
7    integer(kind=IKIND) :: i, j
8    integer, parameter :: rkind = RKIND
9
10   ! This loop will be parallelized
11   do i = 1, szyy
12     YY(i) = 0.0_rkind
13     ! This loop will not be parallelized
14     do j = 1, szhh
15       YY(i) = YY(i) + XX(i + j - 1) * HH(j)
16     enddo
17   enddo
18
19 end subroutine match_filter
```

Parallelization of reduction loops not yet supported, this loop will be serial

Parallelism detection

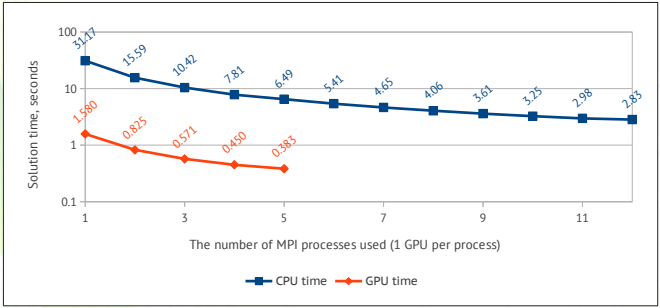
KernelGen analyses loops dependencies and maps parallel loops on GPU compute grid.
Sequential loops are kept unchanged, either inner or outer.

```
1  subroutine match_filter(HH, szhh, XX, szxx, YY, szyy)
2
3    implicit none
4    integer(kind=IKIND), intent(in) :: szhh, szxx, szyy
5    real(kind=RKIND), intent(in) :: HH(szhh), XX(szxx)
6    real(kind=RKIND), intent(out) :: YY(szyy)
7    integer(kind=IKIND) :: i, j
8    integer, parameter :: rkind = RKIND
9
10   ! This loop will be parallelized
11   do i = 1, szyy
12     YY(i) = 0.0_rkind
13     ! This loop will not be parallelized
14     do j = 1, szhh
15       YY(i) = YY(i) + XX(i + j - 1) * HH(j)
16     enddo
17   enddo
18
19 end subroutine match_filter
```

However, the outer loop will be detected parallel and taken on GPU, with inner serial loop processed by each thread

Cooperation with MPI

KernelGen naturally co-exists with MPI parallelism, assigning each MPI process a single GPU:



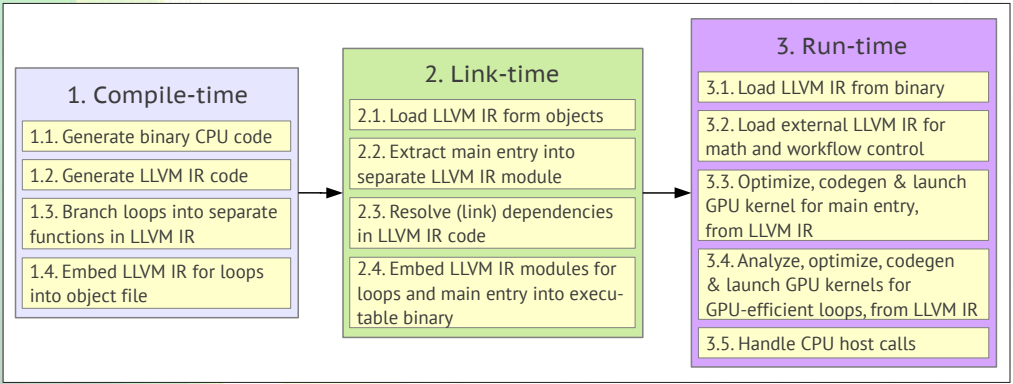
Comparing performance of time domain matched filter application, compiled with KernelGen r1740 for 1-12 cores of Intel Xeon X5670 CPUs and 1-5 NVIDIA Tesla C2070 GPUs (Fermi sm_20)

KernelGen dependencies

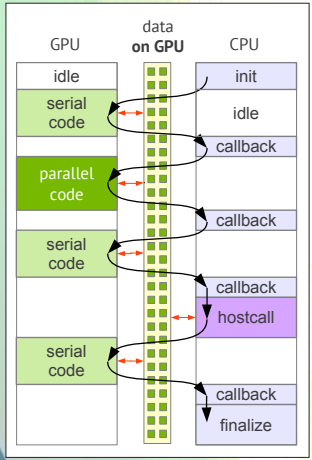
- **GCC** – for frontends and regular compiling pipeline (GPL)
- **DragonEgg** – GCC plugin for converting GCC's IR (gimple) into LLVM IR (GPL)
- **LLVM** – for internal compiler infrastructure (BSD)
- **Polly** – for loops parallelism analysis (BSD+GPL)
- **NVPTX backend** – for emitting LLVM IR into 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)

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.



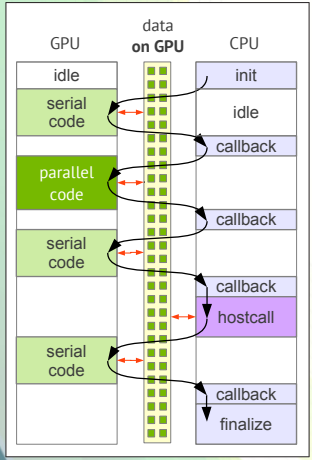
GPU-directed execution



KernelGen uses GPU-directed execution model:

- All activities are initiated by GPU (CPU is “passive”)
- All data (global, dynamic, immediate) resides GPU memory
- GPU data and control flow are managed by *main kernel*, which is persistent on GPU during whole application lifetime

GPU-directed execution

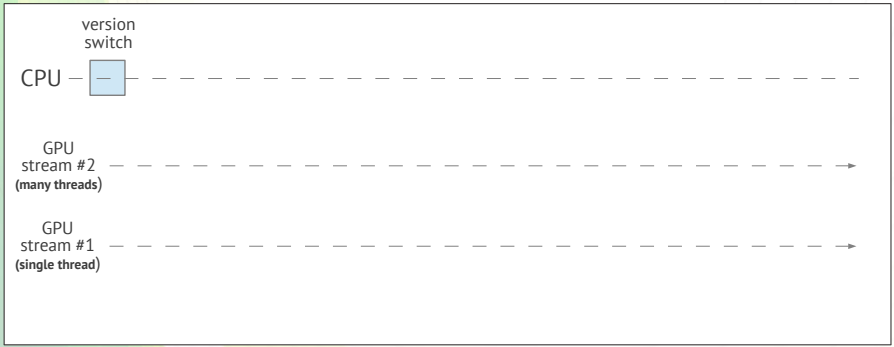


GPU-directed execution model benefits:

- No need to manage host↔device memory transfers explicitly and track hidden data dependencies (side-effects)
- Possibility to utilize long uninterrupted kernels with dynamic parallelism, where available
- Transparent transition of CPU MPI nodes into GPU MPI nodes, with CUDA-aware MPI
- GPU device functions initially loaded with main kernel are shared with all other dynamically loaded kernels, minimizing compilation time and code size (in contrast, OpenACC and CUDA kernels must be self-containing)

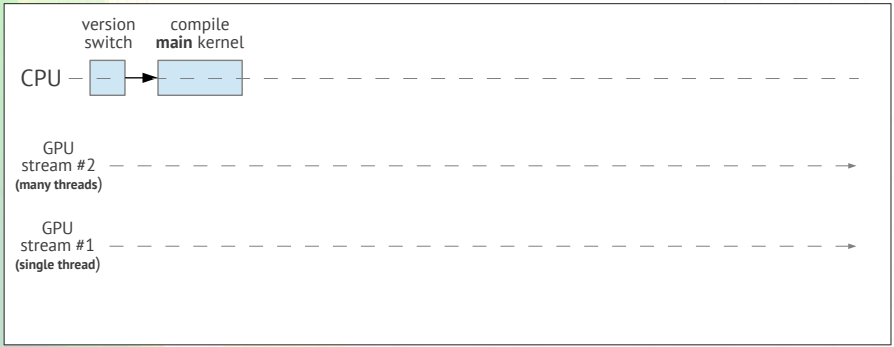
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



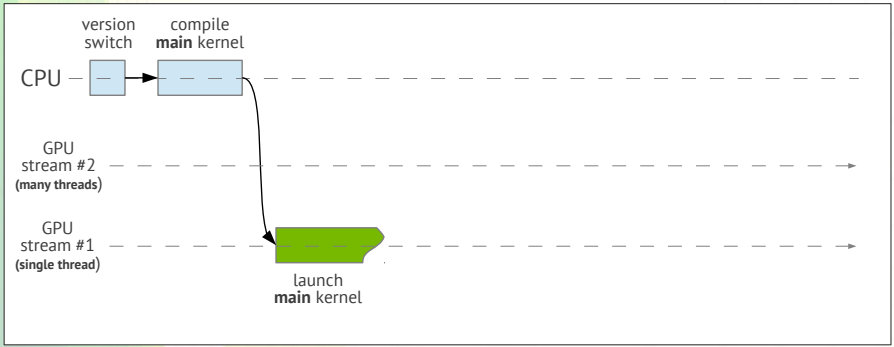
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



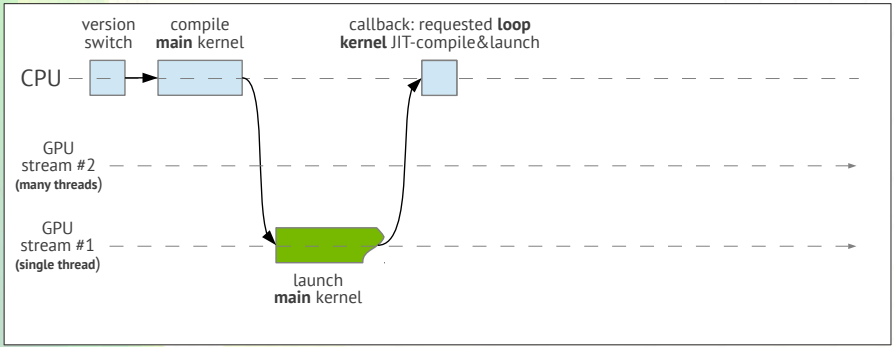
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



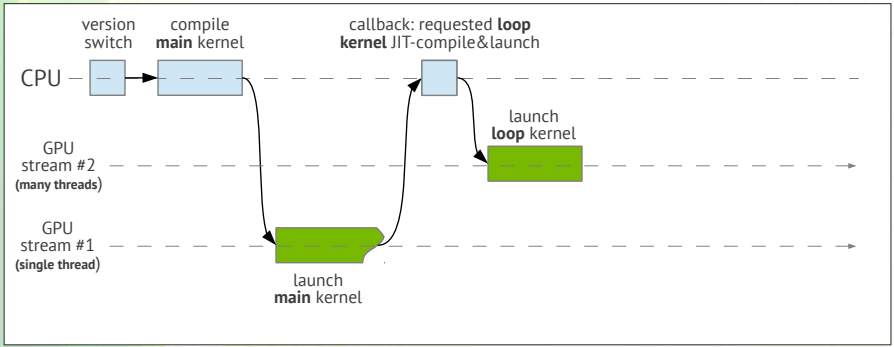
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



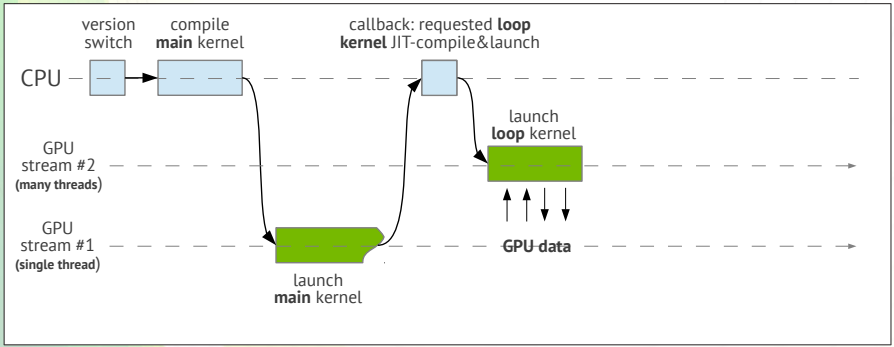
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



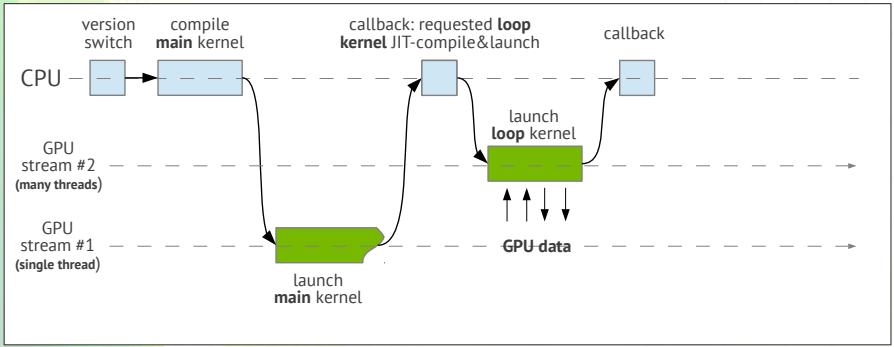
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



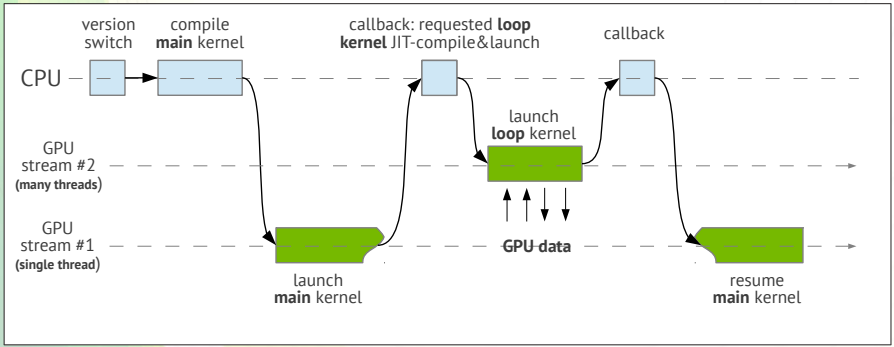
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



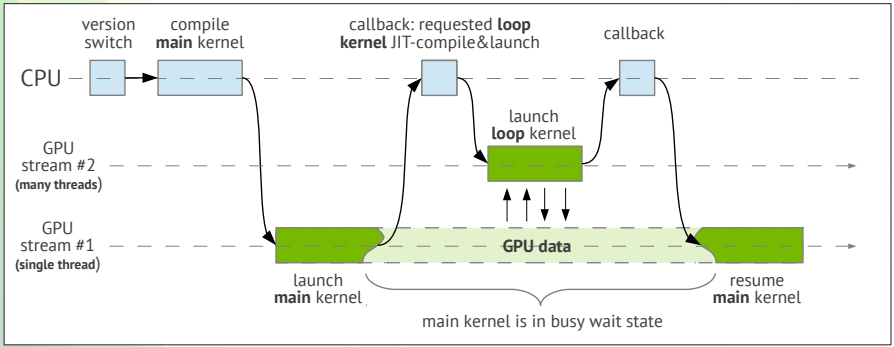
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



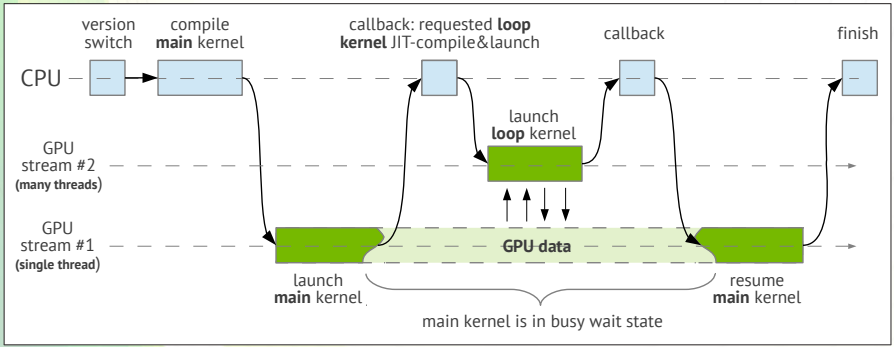
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



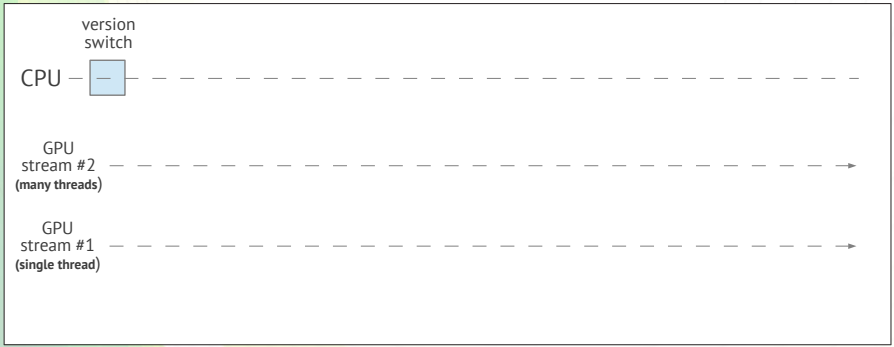
KernelGen execution workflow

Scenario 1: main kernel approached a point where another GPU kernel can be launched (for parallel loop). In this case, another kernel is launched via host callback (could be replaced with dynamic parallelism on Kepler sm_35). All GPU data is shared (*.local* data is disabled).



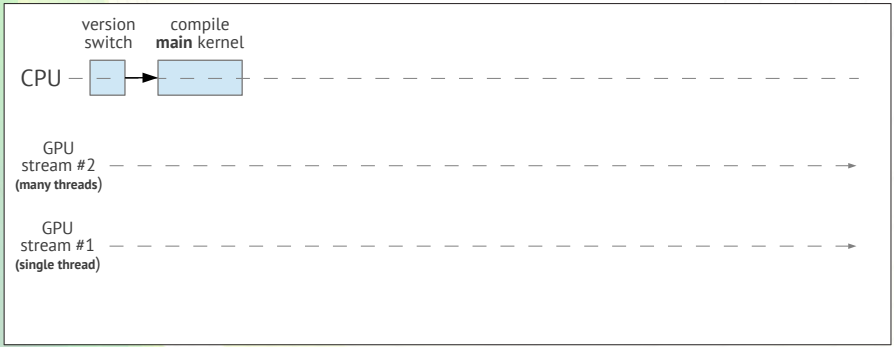
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



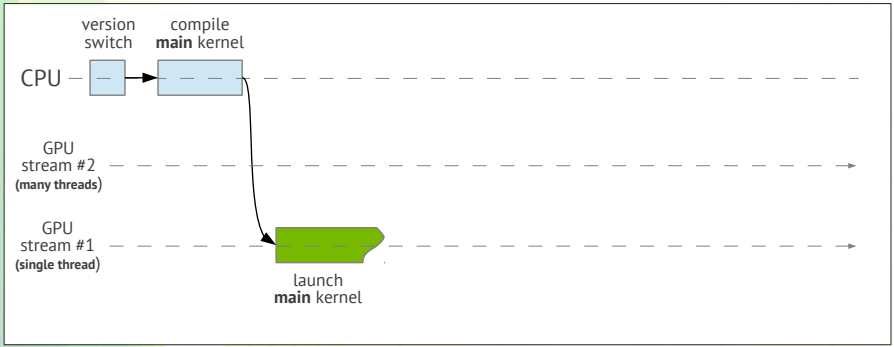
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



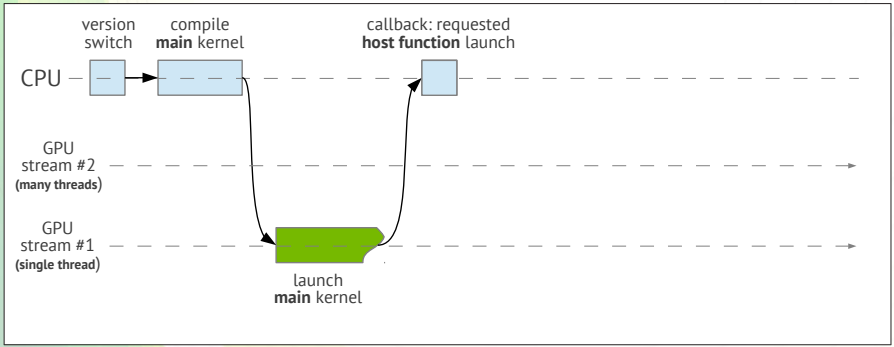
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



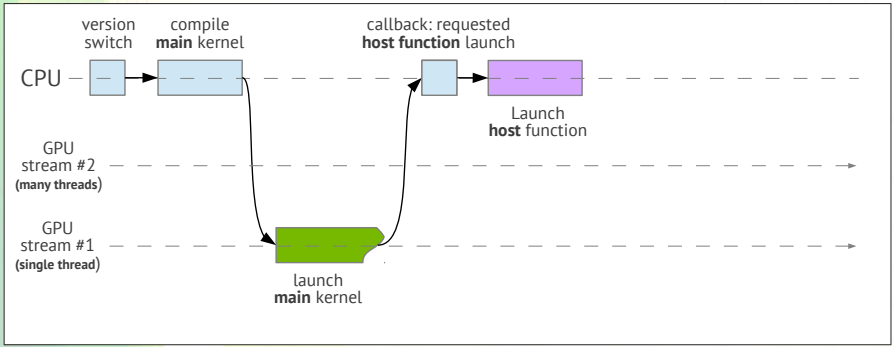
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



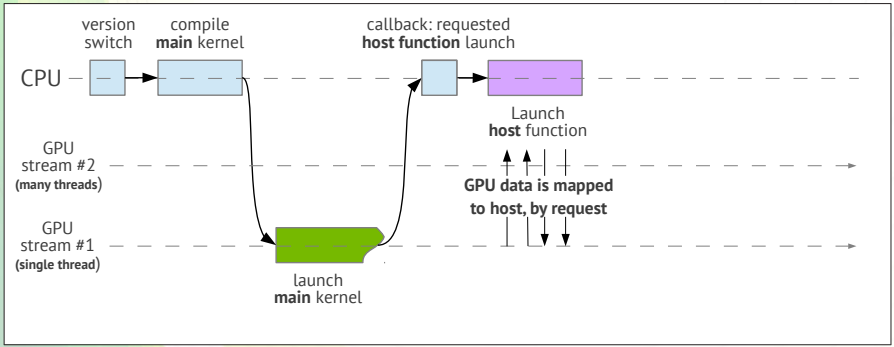
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



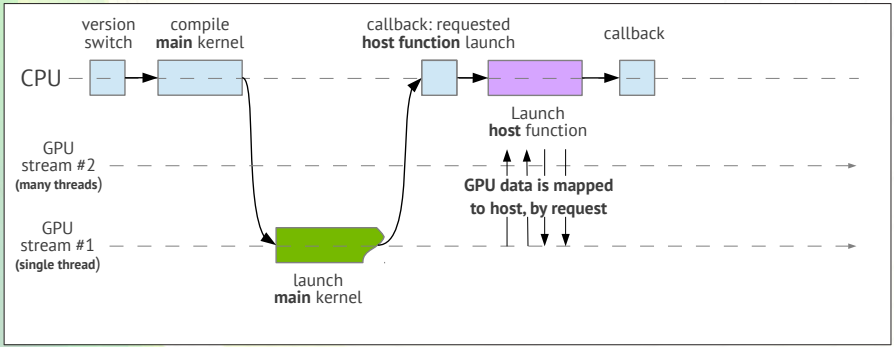
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



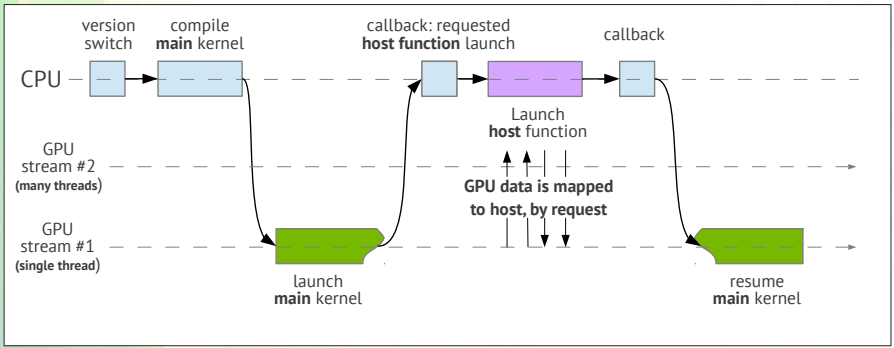
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



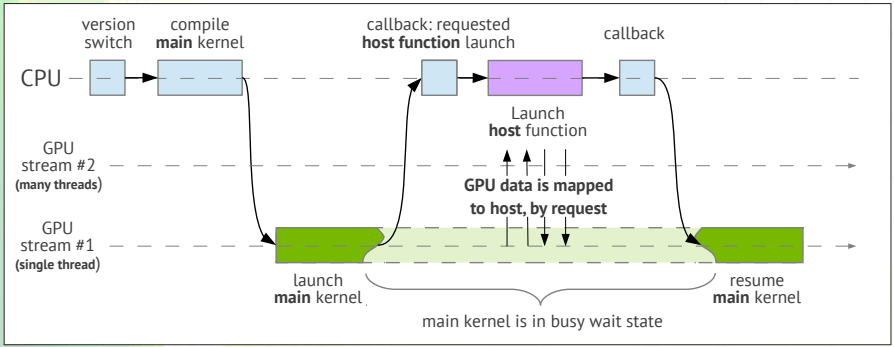
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



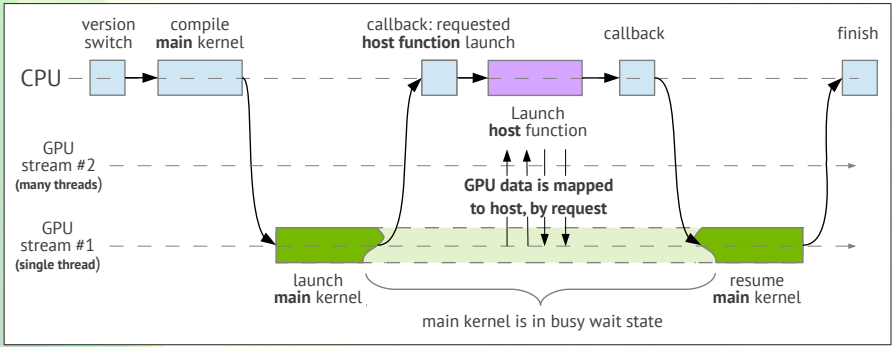
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



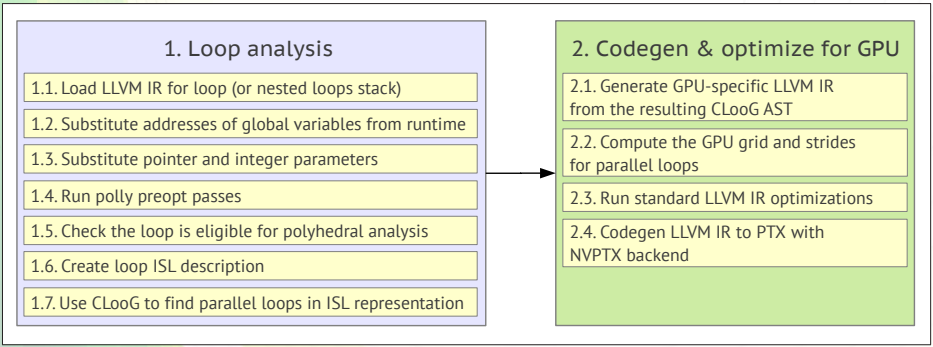
KernelGen execution workflow

Scenario 2: main kernel approached a point where CPU function is called (no source code, external library, syscall, or inefficient loop). In this case, function is called via host callback, using *libffi* API. GPU data is lazily mapped for using on host, where needed, by pagefault handler.



KernelGen loops analysis pipeline

KerenlGen 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.



Code generation brief walk-through

Let's briefly walk through KernelGen code generation steps for the following function:

sincos.c

```
void sincos(int nx, int ny, int nz, float* x, float* y, float* xy) {
    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]);
            }
}
```

Code generation brief walk-through

First, GCC frontend produces GIMPLE IR from any high-level language:

```
1  sincos (integer(kind=4) & restrict nx, integer(kind=4) & restrict ny, integer(kind=4) & restrict nz, real(kind=4)[0:D↩
       .1629] * restrict x, real(kind=4)[0:D.1626] * restrict y, real(kind=4)[0:D.1632] * restrict xy)
2  {
3    ...
4    D.1646 = ~stride.16;
5    offset.19 = D.1646 - stride.18;
6    {
7      ...
8      {
9        ...
10       {
11         if (j <= D.1568) goto <D.1650>; else goto <D.1651>;
12         <D.1650>:
13         <D.1652>:
14         {
15           logical(kind=4) D.1576;
16           {
17             ...
18             {
19               logical(kind=4) D.1575;
20               ...
21               D.1676 = sincos_ijk (D.1675, D.1669);
22               *xy[D.1663] = D.1676;
```

Code generation brief walk-through

Next, DragonEgg converts GIMPLE IR to LLVM IR:

```
; ModuleID = '../sincos.f90'
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"

declare float @sincos_ijk_(...)
...
define void @sincos_(i32* noalias %nx, i32* noalias %ny, i32* noalias %nz, [0 x float]* noalias %x, [0 x float]* noalias %y, [0 x float]* noalias %xy) nounwind uwtable {
  ...
  %92 = add i64 %89, %91
  %93 = add i64 %87, %92
  %94 = add i64 %93, %41
  %95 = bitcast [0 x float]* %4 to float*
  %96 = getelementptr float* %95, i64 %94
  %97 = call float bitcast (float (...)* @sincos_ijk_ to float (float*, float*)*)(float* %96, float* %86) nounwind
  %98 = bitcast [0 x float]* %5 to float*
  %99 = getelementptr float* %98, i64 %76
  store float %97, float* %99, align 4
  %100 = icmp eq i32 %68, %66
  %101 = add i32 %68, 1
  %102 = icmp ne i1 %100, false
  br i1 %102, label %"7", label %"6"
```

Code generation brief walk-through

KernelGen compile-time extracts loops potentially suitable for GPU kernels into new functions.
Groups of multiple loops are extracted recursively, in case only subset of them is parallel.
KernelGen runtime substitutes constants and pointers from runtime context for easier analysis:

```
1  ; ModuleID = '__kernelgen_sincos__loop_3_module'
2  target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64↩
       :64:64-v128:128:128-n16:32:64"
3  target triple = "nvptx64-unknown-unknown"
4  define void @__kernelgen_sincos__loop_3(i32*) nounwind {
5  "Loop Function Root":
6    br label %"4.preheader.cloned"
7    ...
8    %8 = getelementptr [0 x float]* inttoptr (i64 47313862656 to [0 x float]*), i64 0, i64 %7
9    %9 = getelementptr [0 x float]* inttoptr (i64 47246749696 to [0 x float]*), i64 0, i64 %7
10   %10 = load float* %9, align 4
11   %11 = call float @sinf(float %10) nounwind readnone alignstack(1)
12   %12 = load float* %8, align 4
13   %13 = call float @cosf(float %12) nounwind readnone alignstack(1)
14   %14 = fadd float %11, %13
15   %15 = getelementptr [0 x float]* inttoptr (i64 47380975616 to [0 x float]*), i64 0, i64 %7
16   store float %14, float* %15, align 4
17   ...
18 declare float @sinf(float) nounwind readnone alignstack(1)
19 declare float @cosf(float) nounwind readnone alignstack(1)
```

Code generation brief walk-through

KernelGen runtime analyses kernel loop for parallelism. In case it is parallel, LLVM IR is specialized for NVPTX (note *@llvm.nvvm.** intrinsics, *ptx_kernel* and *ptx_device*):

```
; ModuleID = '__kernelgen_sincos__loop_3_module'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:64-v16:16:16-v32:32:32-v64
    :64:64-v128:128:128-n16:32:64"
target triple = "nvptx64-unknown-unknown"

@__kernelgen_version = constant [15 x i8] c"0.2/1654:1675M\00"

define ptx_kernel void @__kernelgen_sincos__loop_3(i32* nocapture) nounwind alwaysinline {
"Loop Function Root":
  %tid.z = tail call ptx_device i32 @llvm.nvvm.read.ptx.sreg.tid.z()
  %ctaid.z = tail call ptx_device i32 @llvm.nvvm.read.ptx.sreg.ctaid.z()
  %PositionOfBlockInGrid.z = shl i32 %ctaid.z, 2
  %BlockLB.Add.ThreadPosInBlock.z = add i32 %PositionOfBlockInGrid.z, %tid.z
  ...
  %p_28 = tail call ptx_device float @sinf(float %_p_scalar_) nounwind readnone alignstack(1)
  %_p_scalar_29 = load float* %p_scevgep6
  %p_30 = tail call ptx_device float @cosf(float %_p_scalar_29) nounwind readnone alignstack(1)
  %p_31 = fadd float %p_28, %p_30
  store float %p_31, float* %p_scevgep
  br label %CUDA.AfterLoop.z
  ...
```

Code generation brief walk-through

PTX assembler is generated from LLVM IR, using LLVM NVPTX backend:

```
.visible .entry __kernelgen_sincos__loop_3(.param .u64 __kernelgen_sincos__loop_3_param_0)
{
    ...
// BB#0:                                // %Loop Function Root
    mov.u32  %r0, %tid.z;
    mov.u32  %r1, %ctaid.z;
    shl.b32  %r1, %r1, 2;
    add.s32  %r2, %r1, %r0;
    @%p0 bra  BB0_4;
    ...
    // Callseq Start 42
    {
    .reg .b32 temp_param_reg;
    // <end>}
    .param .b32 param0;
    st.param.f32 [param0+0], %f0;
    .param .b32 retval0;
    call.uni (retval0),
    sinf,
    (
    param0
    );
    ...
```

Code generation brief walk-through

Finally, a Fermi ISA is generated for function, in no-cloning mode. Unresolved function calls (*JCAL 0x0*) are replaced with actual addresses of functions already loaded by main kernel.

```
Function : __kernelgen_sincos__loop_3
/*0008*/     /*0x10005de428004001*/  MOV R1, c [0x0] [0x44];
/*0010*/     /*0x9c00dc042c000000*/  S2R R3, SR_CTAid_Z;
/*0018*/     /*0x8c001c042c000000*/  S2R R0, SR_Tid_Z;
...
/*0120*/     /*0x08451c036000c000*/  SHL R20, R4, 0x2;
/*0128*/     /*0x14055c4340000000*/  ISCADD R21, R0, R5, 0x2;
/*0130*/     /*0x01411c020c0080c0*/  IADD32I R4.CC, R20, 0x203000;
/*0138*/     /*0x2d515c434800c000*/  IADD.X R5, R21, 0xb;
/*0148*/     /*0x00411c8584000000*/  LD.E R4, [R4];
/*0150*/     /*0x00010007100017b2*/  JCAL 0x5ec80;
/*0158*/     /*0x01419c020c108100*/  IADD32I R6.CC, R20, 0x4204000;
/*0160*/     /*0x10009de428000000*/  MOV R2, R4;
/*0168*/     /*0x2d51dc434800c000*/  IADD.X R7, R21, 0xb;
/*0170*/     /*0x00611c8584000000*/  LD.E R4, [R6];
/*0178*/     /*0x00010007100018eb*/  JCAL 0x63ac0;
/*0188*/     /*0x01419c020c208140*/  IADD32I R6.CC, R20, 0x8205000;
/*0190*/     /*0x10201c0050000000*/  FADD R0, R2, R4;
/*0198*/     /*0x2d51dc434800c000*/  IADD.X R7, R21, 0xb;
/*01a0*/     /*0x00601c8594000000*/  ST.E [R6], R0;
/*01a8*/     /*0x00001de780000000*/  EXIT;
```

KernelGen current limitations

KernelGen still has several important features not covered:

- Loop index must be incremented with positive unit value
- No support for arbitrary indirect indexing, e.g. $a(b(i))$ (same issue in OpenACC)
- No support for reduction idiom (supported by OpenACC)
- Polly may face insufficient code optimization for proper parallelizing some specific test cases

Conclusions

1. KernelGen project implemented a full compiler prototype to produce parallel GPU kernels from unmodified CPU source code
2. C and Fortran languages support is widely tested, focusing mostly on Fortran
3. Performance of GPU kernels generated by KernelGen is 0.5x-4x in comparison to newest commercial PGI OpenACC compiler
4. KernelGen eases porting of applications on GPU by means of supporting many features missing in OpenACC
5. GPU-directed execution model allows seamless transition between CPU-MPI and GPU-MPI versions of application
6. KernelGen is based on freeware and mostly open-source technologies

How to test?

- **Note you will need SM_20 or newer NVIDIA GPU and CUDA 5.0 driver installed**
- Compile KernelGen for your target system, as described in guide
- Use KernelGen to compile a program with computational loops:

```
$ source ~/rpmbuild/CHROOT/opt/kernelgen/usr/bin/kernelgen-vars.sh
$ kernelgen-gfortran test.f90 -o test
```

- Deploy the code on GPU and check out the verbose reports about launched GPU kernels:

```
$ kernelgen_runmode=1 ./test <args>
```

- Compare performance and result to the original CPU version:

```
$ ./test <args>
```

- **Encountered an issue? Please file it into our public bug tracker**

Testing COSMO and WRF

KernelGen is able to compile original source code for COSMO and WRF into valid executables. Both models are now being actively tested.

WRF:

```
$ 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
```

Testing COSMO and WRF

KernelGen is able to compile original source code for COSMO and WRF into valid executables. Both models are now being actively tested.

COSMO:

```
$ source ~/rpmbuild/CHROOT_release/opt/kernelgen/usr/bin/kernelgen-vars.sh
$ ./setup
Select build configuration:
1. gnu
2. kernelgen
3. path64
Enter the value: 2
Configured build for kernelgen
...
$ OMPI_FC=kernelgen-gfortran OMPI_CC=kernelgen-gcc NETCDF=~/rpmbuild/CHROOT_release/opt/kernelgen/usr/ make -j12
...
$ cd ../tests/
$ tar -xjf 2006102312.tar.bz2
$ cd 2006102312/
$ COSMO_NPX=1 COSMO_NPY=8 COSMO_NPIO=0 ./run_eu
```

Collaboration proposal

KernelGen team is seeking for opportunities to:

1. Help research groups to evaluate KernelGen in different applications on GPU-enabled HPC facilities
2. Encourage contributions to existing development plan (tiling, parallelism detection in sparse data algorithms, and many more) and new proposals involving LLVM, Polly/CLooG, NVPTX and AsFermi
3. Offer entire open-source GPU compiler framework as a unified platform for compiler optimizations research

It is also possible to use KernelGen as a base for OpenACC compiler development!

KernelGen

Download link for this presentation:
http://kernelgen.org/gtc2013/

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

Thank you!

HP2C

Università della Svizzera italiana

Facoltà di scienze informatiche