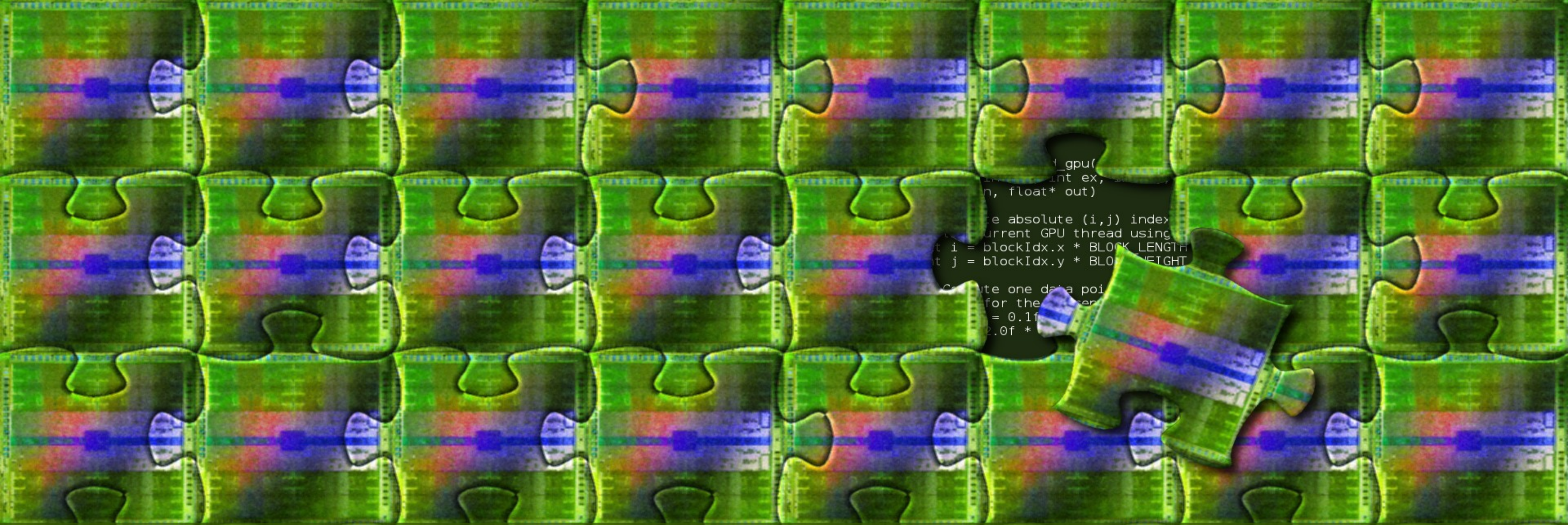


KernelGen

naïve CUDA kernels generation from Fortran source code

Dmitry Mikushin

Contents

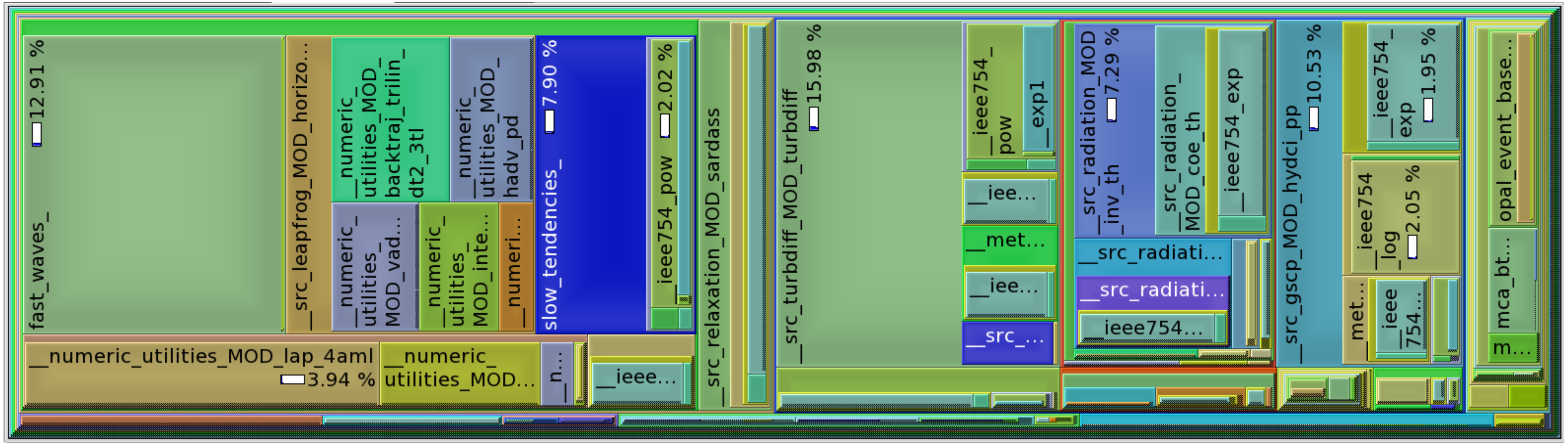
- Motivation, target, analysis
- Assembling our own toolchain
- Toolchain usecase: axpy example
- Development schedule

1. Motivation, target, analysis

Why generation?

The need of huge numerical models porting onto GPUs:

- All individual model blocks have too small self perf impact (~10%), resulting into small speedups, if only one block is ported



Why generation?

The need of huge numerical models porting onto GPUs:

- A lot of code requiring lots of similar transformations
- A lot of code versions with minor differences, each requiring manual testing & support
- COSMO, Meteo-France: science teams are not ready to work with new paradigms (moreover, tied with propriety products), compute teams have no resources to support a lot of new code

Why generation?

So, in fact customers are ready to start GPU-based modeling, if three main requirements are met:

- Model works on GPUs, but conserves the original CPU source code
- Model works on GPUs and gives accurate enough results in comparison with control host version
- Model works on GPUs faster

Our target

Port already parallel models in Fortran onto GPUs:

- Conserving original Fortran source code (i.e. keeping all C/CUDA in intermediate files)
- Minimizing manual work on specific code (i.e. developed toolchain is expected to be reusable with other codes)

“Already parallel” means the model gives us some decomposition (e.g. [istart .. iend] × [jstart .. jend]), that could be used with multiple GPUs as well.

2011

Similar tools: PGI CUDA Fortran

Not really similar:

- Same manual coding as with CUDA C we want to minimize
- PGI's own Fortran language extensions

Similar tools: PGI Accelerator

Very similar, but:

- Still needs to set manual annotations on loops
- Is a propriety “black box” with limited info about implemented features

Similar tools: PathScale HMPP

Almost same as PGI Accelerator:

- Also has CAPS for automatic capable loops lookup
- Introduces some inapplicable constraints on accelerated loops, for instance, being a pure function (not really a problem with partial link-time kernels compilation)
- Tries to standardize OpenHMPP
- HMPP is a “black box”, but PathScale host compiler recently became open-source

Similar tools: f2c-acc

Actually, a work-in-progress equivalent for PGI Accelerator

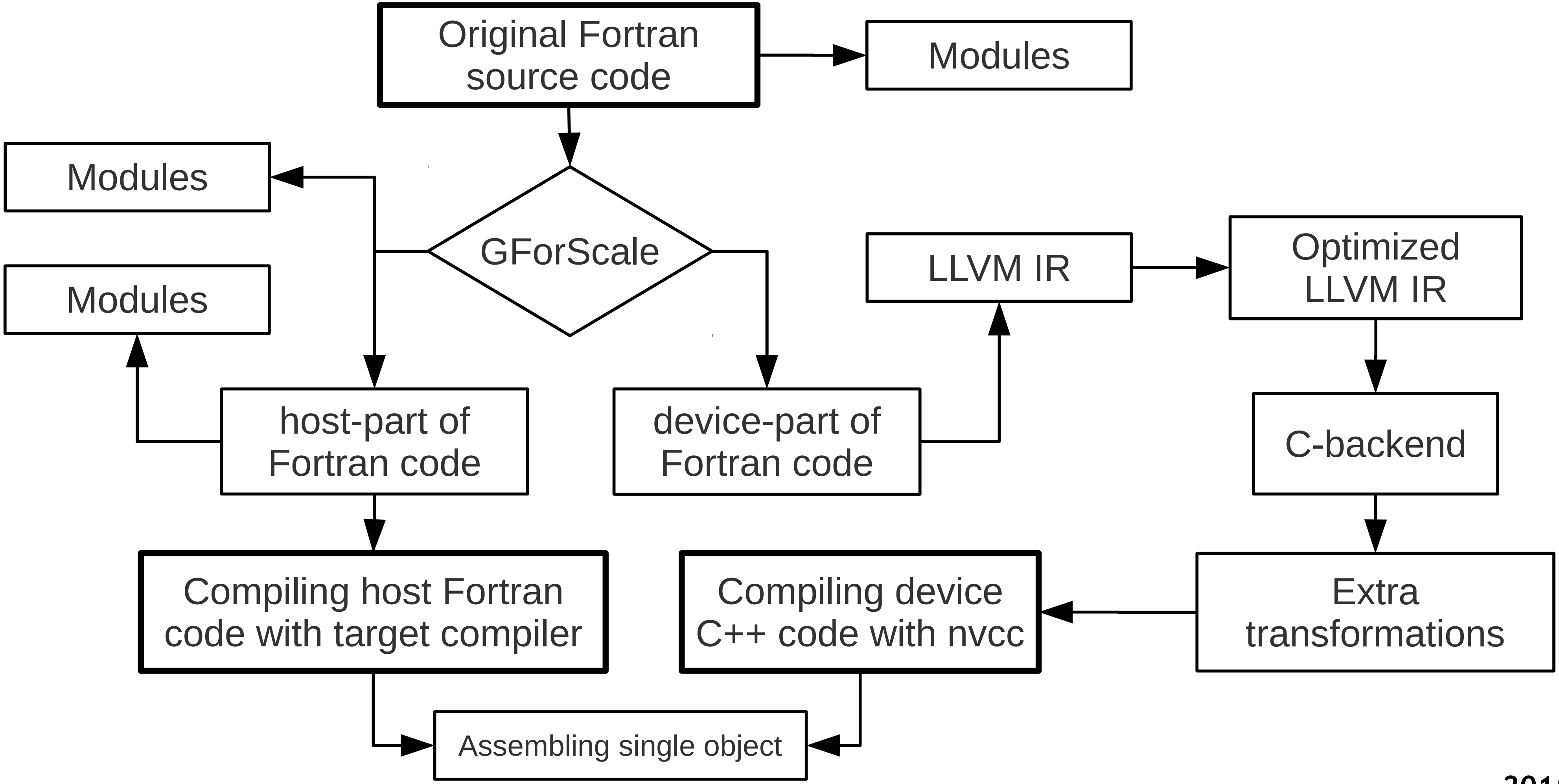
- Still a lot of manual assistance needed

Conclusion

- Clearly, it would be a right decision to use f2c-acc or PGI or HMPP, if they could support GPU kernels generation without directives/annotations
- While they don't, adopting models is a complicated long-term task
- Can we build our own toolchain with dependencies analysis instead of directives?

2. Assembling our own toolchain

Toolchain workflow



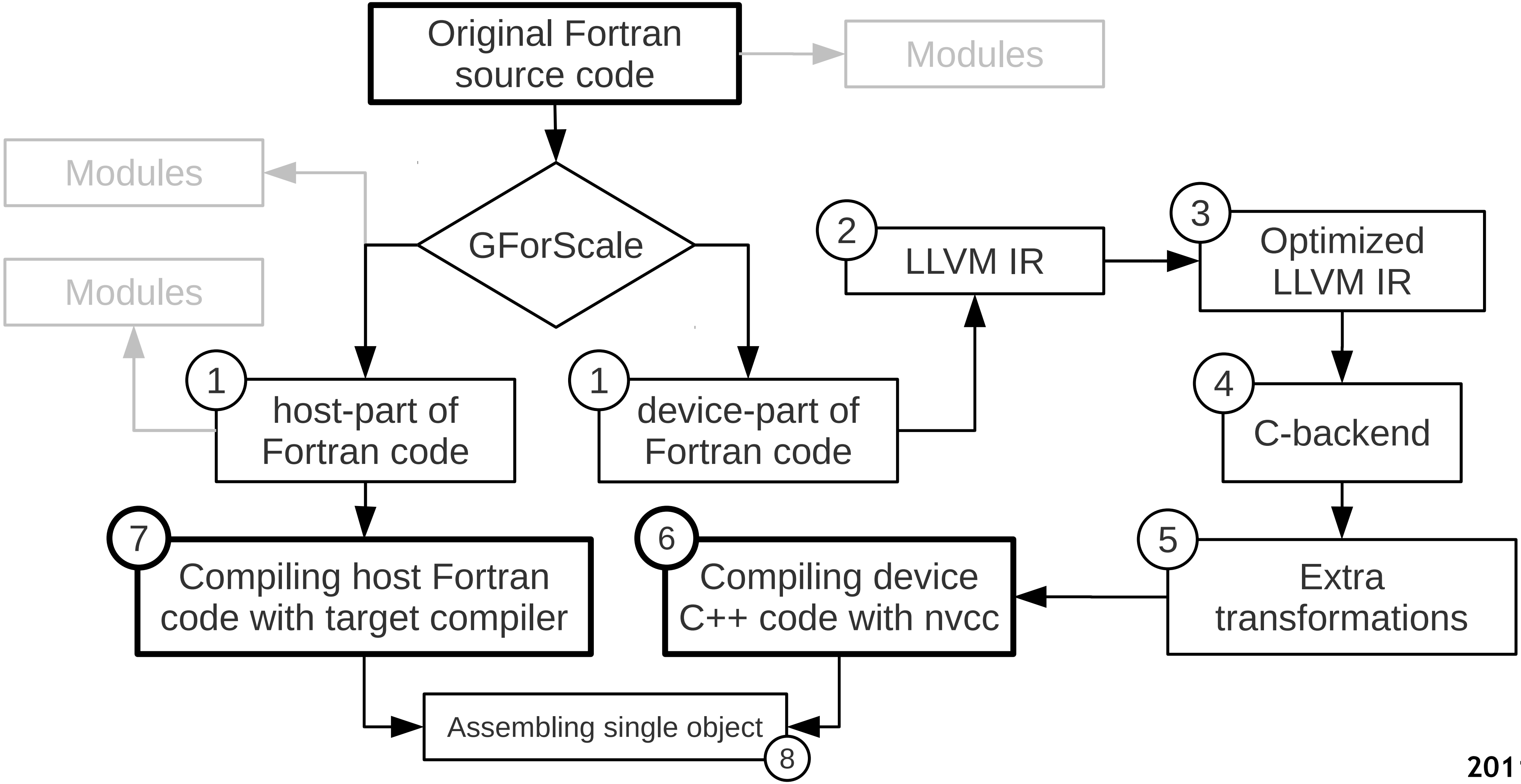
2011

Toolchain installation

```
/opt/kgen/
    bin/
        g95xml-refids
        g95xml-tree                    } Source code XML-markup
        kgen                           } Script performing steps 1-8
        kgen-gforscale                 } Source-to-source translator
        kgen-gfortran                  } Frontend ("compiler")
    include/
        gforscale.h
        gforscale.mod
        gforscale.dragonegg.mod        } Runtime-modules
    lib/
        libgforscale.so                } Runtime-library
    transforms/
        split/                         } Tree of code transformation
                                         rules performing host/device
                                         code split
```

3. Toolchain usecase

8 basic steps in total



Example: axpy

Consider toolchain steps in detail for the following simple test program:

```
subroutine axpy(n, a, x, y)

implicit none

integer, intent(in) :: n
real, intent(in) :: a, x(n)
real, intent(inout) :: y(n)

integer :: i

do i = 1, n
  y(i) = y(i) + a * x(i)
enddo

print *, 'Value of i after cycle = ', i

end subroutine axpy
```

Step 1

kgen-gforscale -Wk,--gforscale-scene-path=/opt/kgen/transforms/split/ -Wk,--gforscale-mode=tree axpy.f90

```
subroutine axpy_loop_1_gforscale(n, y, a, x)
implicit none
interface
subroutine axpy_loop_1_gforscale_blockidx_x(index, start, end)
bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
subroutine axpy_loop_1_gforscale_blockidx_y(index, start, end)
bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
end interface
integer :: i
integer, intent(in) :: n
real, intent(inout) :: y(n)
real, intent(in) :: a
real, intent(in) :: x(n)

#ifdef __CUDA_DEVICE_FUNC__
call axpy_loop_1_gforscale_blockidx_x(i, 1, n)
#else
do i = 1, n
#endif

  y(i) = y(i) + a * x(i)

#ifndef __CUDA_DEVICE_FUNC__
enddo
#endif

end subroutine axpy_loop_1_gforscale
```

device-code

```
subroutine axpy(n, a, x, y)
USE GFORSCALE


implicit none

integer, intent(in) :: n
real, intent(in) :: a, x(n)
real, intent(inout) :: y(n)

integer :: i


!$GFORSCALE SELECT axpy_loop_1_gforscale
if (gforscale_select(0, 1, 'axpy' // char(0))) then
!$GFORSCALE CALL axpy_loop_1_gforscale
#ifdef __CUDA_DEVICE_FUNC__
  call gforscale_launch('axpy_loop_1_gforscale_' // char(0), &
    1, n, 0, 0, 4, n, sizeof(n), y, sizeof(y), a, sizeof(a), x, sizeof(x))
i = n + 1
#else
  call axpy_loop_1_gforscale(n, y, a, x)
#endif
!$GFORSCALE END CALL axpy_loop_1_gforscale
else
!$GFORSCALE LOOP axpy_loop_1_gforscale
do i = 1, n
  y(i) = y(i) + a * x(i)
enddo
!$GFORSCALE END LOOP axpy_loop_1_gforscale
endif
!$GFORSCALE END SELECT axpy_loop_1_gforscale

print *, 'Value of i after cycle = ', i

end subroutine axpy
```

host-code

Step 2

dragonegg-gfortran -c axpy.axpy_loop_1_gforscale.F90 -D__CUDA_DEVICE_FUNC__ -ffree-line-length-none -fplugin=/opt/llvm/dragonegg/lib64/dragonegg.so -O0 -S -fplugin-arg-dragonegg-emit-ir -o axpy.axpy_loop_1_gforscale.F90.bc

```
; ModuleID = 'axpy.axpy_loop_1_gforscale.F90'
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-v64:64:64-v128:128:128-a0:0:64-s0:64:64-
f80:128:128-f128:128:128-n8:16:32:64"
target triple = "x86_64-unknown-linux-gnu"

module asm "\09.ident\09\22GCC: (GNU) 4.5.4 20110527 (prerelease) LLVM:
131968\22"

%"integer(kind=4)" = type i32
%"real(kind=4)" = type float

define void @axpy_loop_1_gforscale_(i32* %n, [0 x float]* %y, float* %a, [0
x float]* %x) nounwind {
entry:
  %n_addr = alloca i32*, align 8
  %y_addr = alloca [0 x float]*, align 8
  %a_addr = alloca float*, align 8
  %x_addr = alloca [0 x float]*, align 8
  %memtmp = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  store i32* %n, i32** %n_addr
  store [0 x float]* %y, [0 x float]** %y_addr
  store float* %a, float** %a_addr
  store [0 x float]* %x, [0 x float]** %x_addr
  %0 = load i32** %n_addr, align 64
  %1 = load [0 x float]** %y_addr, align 64
  %2 = load float** %a_addr, align 64
  %3 = load [0 x float]** %x_addr, align 64
  %"ssa point" = bitcast i32 0 to i32
  br label %"2"

"2":                                              ; preds = %entry
  %4 = load i32* %0, align 4
  %5 = sext i32 %4 to i64
  %6 = icmp sge i64 %5, 0
  %7 = select i1 %6, i64 %5, i64 0
  %8 = add nsw i64 %7, -1
  %9 = mul i64 %7, 32
  %10 = mul i64 %7, 4
  %11 = load i32* %0, align 4
  %12 = sext i32 %11 to i64
  %13 = icmp sge i64 %12, 0
  %14 = select i1 %13, i64 %12, i64 0
  %15 = add nsw i64 %14, -1
  %16 = mul i64 %14, 32
  %17 = mul i64 %14, 4
  %18 = load i32* %0, align 4
  call void (i32*, i32, i32, ...)* @axpy_loop_1_gforscale_blockidx_x(i32*
noalias %memtmp, i32 1, i32 %18) nounwind
  %19 = load i32* %memtmp, align 4
  %20 = sext i32 %19 to i64
  %21 = add nsw i64 %20, -1
  %22 = load i32* %memtmp, align 4
  %23 = sext i32 %22 to i64
  %24 = add nsw i64 %23, -1
  %25 = bitcast [0 x float]* %1 to float*
  %26 = getelementptr float* %25, i64 %24
  %27 = load float* %26, align 4
  %28 = load float* %2, align 4
  %29 = load i32* %memtmp, align 4
  %30 = sext i32 %29 to i64
  %31 = add nsw i64 %30, -1
  %32 = bitcast [0 x float]* %3 to float*
  %33 = getelementptr float* %32, i64 %31
  %34 = load float* %33, align 4
  %35 = fmul float %28, %34
  %36 = fadd float %27, %35
  %37 = bitcast [0 x float]* %1 to float*
  %38 = getelementptr float* %37, i64 %21
  store float %36, float* %38, align 4
  br label %return

return:                                           ; preds = %"2"
  ret void
}

declare void @axpy_loop_1_gforscale_blockidx_x(i32* noalias, i32, i32, ...)
```

Step 3

/opt/llvm/bin/opt -std-compile-opts axpy.axpy_loop_1_gforscale.F90.bc -S -o axpy.axpy_loop_1_gforscale.F90.bc.opt

```
; ModuleID = 'axpy.axpy_loop_1_gforscale.F90.bc'
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-v64:64:64-v128:128:128-a0:0:64-
s0:64:64-f80:128:128-f128:128:128-n8:16:32:64"
target triple = "x86_64-unknown-linux-gnu"

module asm "\09.ident\09\22GCC: (GNU) 4.5.4 20110527 (prerelease) LLVM: 131968\22"

define void @axpy_loop_1_gforscale_(i32* nocapture %n, [0 x float]* nocapture %y, float* %a, [0 x float]* %x) nounwind {
entry:
  %memtmp = alloca i32, align 4
  %0 = load i32* %n, align 4
  call void (i32*, i32, i32, ...)* @axpy_loop_1_gforscale_blockidx_x(i32* noalias %memtmp, i32 1, i32 %0) nounwind
  %1 = load i32* %memtmp, align 4
  %2 = sext i32 %1 to i64
  %3 = add nsw i64 %2, -1
  %4 = getelementptr [0 x float]* %y, i64 0, i64 %3
  %5 = load float* %4, align 4
  %6 = load float* %a, align 4
  %7 = getelementptr [0 x float]* %x, i64 0, i64 %3
  %8 = load float* %7, align 4
  %9 = fmul float %6, %8
  %10 = fadd float %5, %9
  store float %10, float* %4, align 4
  ret void
}

declare void @axpy_loop_1_gforscale_blockidx_x(i32* noalias, i32, i32, ...)
```

Step 4

/opt/llvm/bin/llc -march=c axpy.axpy_loop_1_gforscale.F90.bc.opt -o axpy.axpy_loop_1_gforscale.F90.bc.cu

```
asm("\t.ident\t\"GCC: (GNU) 4.5.4 20110527 (prerelease) LLVM: 131968\"\n"
"");

...

#ifdef __CUDA_DEVICE_FUNC__
__device__
#endif
void axpy_loop_1_gforscale_(unsigned int *llvm_cbe_n, struct l_unnamed0 (*llvm_cbe_y), float *llvm_cbe_a, struct l_unnamed0 (*llvm_cbe_x));
#ifdef __CUDA_DEVICE_FUNC__
__device__
#endif
void axpy_loop_1_gforscale_blockidx_x(unsigned int *, unsigned int , unsigned int );

void axpy_loop_1_gforscale_(unsigned int *llvm_cbe_n, struct l_unnamed0 (*llvm_cbe_y), float *llvm_cbe_a, struct l_unnamed0 (*llvm_cbe_x)) {
  unsigned int llvm_cbe_memtmp;    /* Address-exposed local */
  unsigned int llvm_cbe_tmp__1;
  unsigned int llvm_cbe_tmp__2;
  unsigned long long llvm_cbe_tmp__3;
  float *llvm_cbe_tmp__4;
  float llvm_cbe_tmp__5;
  float llvm_cbe_tmp__6;
  float llvm_cbe_tmp__7;

  llvm_cbe_tmp__1 = *llvm_cbe_n;
  axpy_loop_1_gforscale_blockidx_x((&llvm_cbe_memtmp), 1u, llvm_cbe_tmp__1);
  llvm_cbe_tmp__2 = *(&llvm_cbe_memtmp);
  llvm_cbe_tmp__3 = ((unsigned long long )(((unsigned long long )(((signed long long )(signed int )llvm_cbe_tmp__2))) + ((unsigned long long )
18446744073709551615ull)));
  llvm_cbe_tmp__4 = (&(*llvm_cbe_y).array[((signed long long )llvm_cbe_tmp__3)]);
  llvm_cbe_tmp__5 = *llvm_cbe_tmp__4;
  llvm_cbe_tmp__6 = *llvm_cbe_a;
  llvm_cbe_tmp__7 = *((&(*llvm_cbe_x).array[((signed long long )llvm_cbe_tmp__3)]));
  *llvm_cbe_tmp__4 = (((float )(llvm_cbe_tmp__5 + (((float )(llvm_cbe_tmp__6 * llvm_cbe_tmp__7))))));
  return;
}
```

Step 5

```
asm("\t.ident\t\"GCC: (GNU) 4.5.4 20110527 (prerelease) LLVM: 131968 \"\n"
"");

...

#ifdef __CUDA_DEVICE_FUNC__
extern "C" __global__
#endif
void axpy_loop_1_gforscale_( unsigned int *llvm_cbe_n, struct l_unnamed0 (*llvm_cbe_y), float *llvm_cbe_a, struct l_unnamed0 (*llvm_cbe_x));
#ifdef __CUDA_DEVICE_FUNC__
__device__
#endif
void axpy_loop_1_gforscale_blockidx_x( unsigned int* index, unsigned int start, unsigned int end) { *index = blockIdx.x + start; }

void axpy_loop_1_gforscale_( unsigned int *llvm_cbe_n, struct l_unnamed0 (*llvm_cbe_y), float *llvm_cbe_a, struct l_unnamed0 (*llvm_cbe_x)) {
  unsigned int llvm_cbe_memtmp;    /* Address-exposed local */
  unsigned int llvm_cbe_tmp__1;
  unsigned int llvm_cbe_tmp__2;
  unsigned long long llvm_cbe_tmp__3;
  float *llvm_cbe_tmp__4;
  float llvm_cbe_tmp__5;
  float llvm_cbe_tmp__6;
  float llvm_cbe_tmp__7;

  llvm_cbe_tmp__1 = *llvm_cbe_n;
  axpy_loop_1_gforscale_blockidx_x((&llvm_cbe_memtmp),  1u, llvm_cbe_tmp__1);
  llvm_cbe_tmp__2 = *(&llvm_cbe_memtmp);
  llvm_cbe_tmp__3 = (( unsigned long long )((( unsigned long long )((( signed long long )( signed int )llvm_cbe_tmp__2))) + (( unsigned long long )
18446744073709551615ull )));
  llvm_cbe_tmp__4 = (&(*llvm_cbe_y).array[(( signed long long )llvm_cbe_tmp__3)]);
  llvm_cbe_tmp__5 = *llvm_cbe_tmp__4;
  llvm_cbe_tmp__6 = *llvm_cbe_a;
  llvm_cbe_tmp__7 = *((&(*llvm_cbe_x).array[(( signed long long )llvm_cbe_tmp__3)]));
  *llvm_cbe_tmp__4 = ((( float )(llvm_cbe_tmp__5 + ((( float )(llvm_cbe_tmp__6 * llvm_cbe_tmp__7))))));
  return;
}
```

2011

Steps 6-8

Final compilation of host and device parts, assembling into single object file:

```
# 6
nvcc -g -c axpy.axpy_loop_1_gforscale.F90.bc.cu -D__CUDA_DEVICE_FUNC__ -G -o
axpy.axpy_loop_1_gforscale.F90.o

# 7
gfortran -g -c axpy.host.F90 -D__CUDA_DEVICE_FUNC__ -ffree-line-length-none
-I/opt/kgen/include -o axpy.host.F90.o

# 8
/usr/bin/ld --unresolved-symbols=ignore-all -r -o axpy.o_kgen axpy.host.F90.o
axpy.axpy_loop_1_gforscale.F90.o
```

Testing axpy

 # By default – execute on CPU

```
[marcusmae@T61p axpy]$ ./axpy
Usage: ./axpy <n> <eps>
[marcusmae@T61p axpy]$ ./axpy 10 0.001
 Value of i after cycle =           11
Max abs diff = 0.000000
```

 # Use GPU kernel if the corresponding environment variable set to 1
 # (extra debug output showing how arguments are mapped into device memory)

```
[marcusmae@T61p axpy]$ axpy_1=1 ./axpy 1000 0.001
arg 0 maps memory segment [140735344500736 .. 140735344508928] to
[1052672 .. 1060864]
arg 1 maps memory segment [28172288 .. 28184576] to [1060864 .. 1073152]
arg 2 reuses mapping created by arg 0
arg 3 reuses mapping created by arg 1
 Value of i after cycle =         1001
Max abs diff = 0.000000
```

4. Development schedule

Stage 1 (April - June)

- Put together all necessary toolchain parts, write the main script
- Test C code generation, file bugs to llvm, patch C backend for CUDA support
- [variant, taken] Complete existing host-device code split transform (previously started in 2009 for CellBE)
- [variant, not taken] Use ROSE parser & AST to perform host-device code split
- Implement kernel invocation runtime:
  - Kernel launching & arguments packing using CUDA API
  - Automatic data synchronization using memory mapping (zero-copy)
  - Support synchronization for allocatable arrays and data in modules
- Implement kernel self-checking runtime:
  - Add kernel invocation mode with input/output data duplicating
  - For each kernel generate routine computing differences with control results and invoke it after every kernel launch, when self-check enabled
- Compile COSMO with toolchain and present charts showing the percentage of successfully generated kernels with checked correct results

Stage 2 (July - October)

- Improve support/coverage
  - More testing on COSMO and other models, file bugs (+2 RHM fellows)
  - Fix the most hot bugs in host-device code split transform
  - Use Polly/Pluto for more accurate capable loops recognition
  - Support link-time generation for kernels with external dependencies
- Improve efficiency
  - Use shared memory in stencils (+1 contractor)
  - Implement both zero-copy and active data synchronization modes
  - Kernel invocation configs caching
  - [variant] Consider putting serial code into single GPU thread as well, to have the whole model instance running on GPU
  - [variant] Consider selective/prioritized data synchronization support, using data dependencies lookup
  - [variant, suggested by S.K.] CPU ↔ GPU work sharing inside MPI process
- Compare performance with other generation tools
- Present the work and <u>carefully listen to feedback</u>