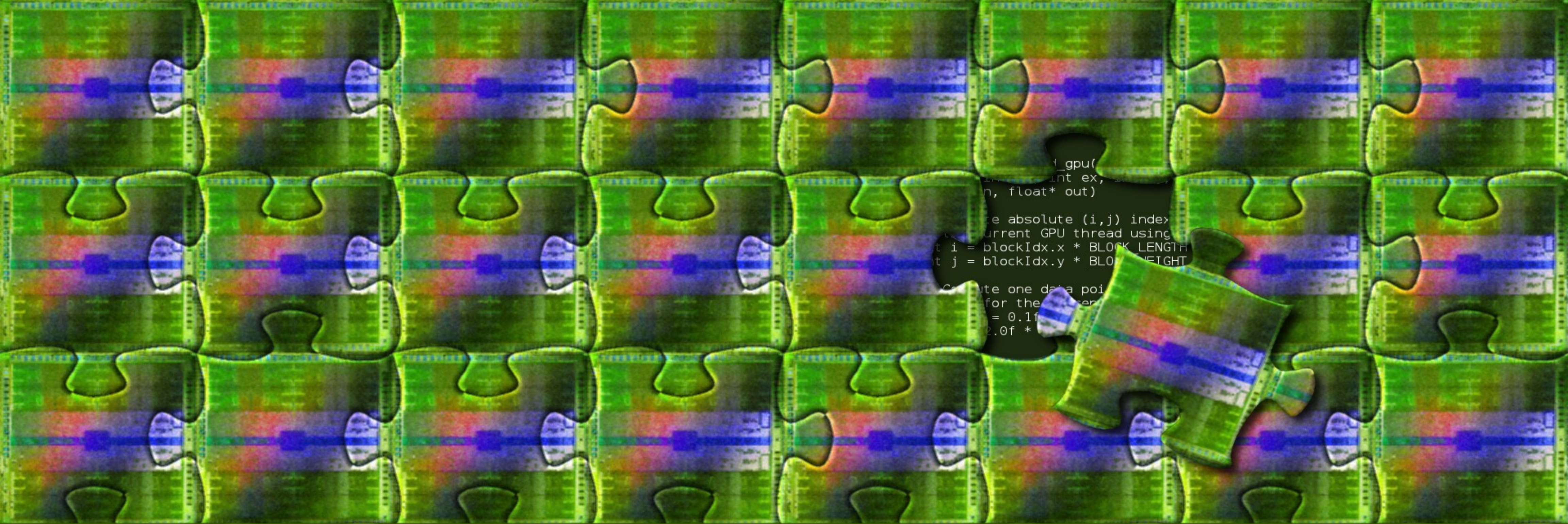


KernelGen naïve GPU kernels generation from Fortran source code

Dmitry Mikushin

Contents

- Motivation and target
- Assembling our own toolchain: schemes and details
- Toolchain usecase: sincos example
- Development schedule

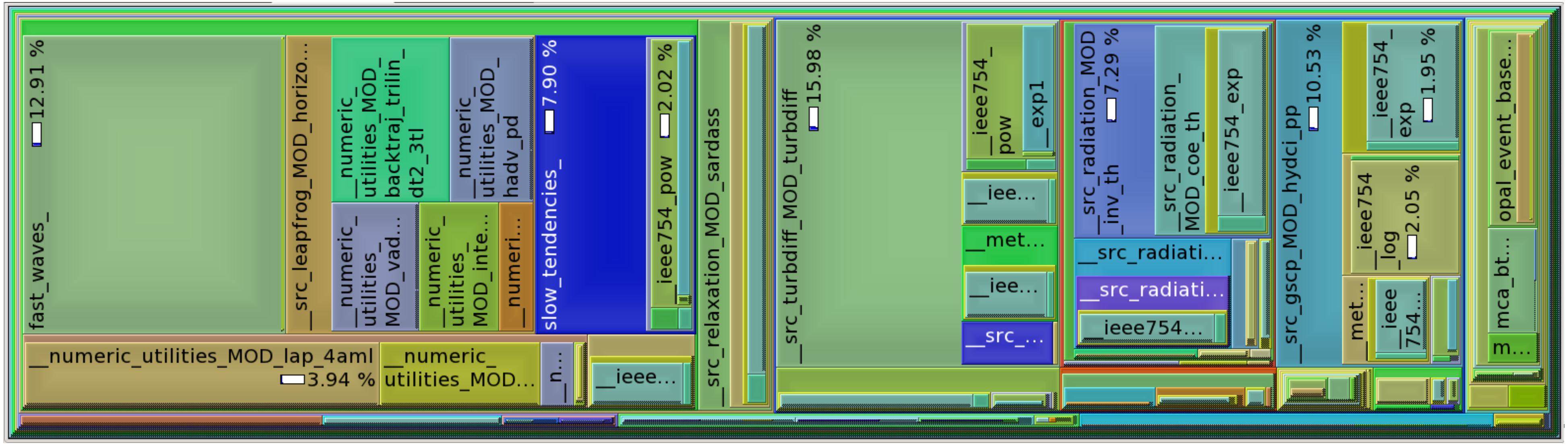
1. Motivation and target

2011

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

2011

Why generation?

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

- Model works on GPUs without specific extensions
- 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/OpenCL 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 data decomposition grid to map 1 GPU onto 1 MPI process or thread.

Similar tools

- PGI CUDA Fortran, Accelerator
- (Open)HMPP by CAPS and Pathscale
- f2c-acc

Common weaknesses: manual coding, proprietary, non-standard, non-free, closed source, non-customizable, etc.

Although, pros & cons of these toolchains is a long discussion omitted here.

2. Assembling our own toolchain

Ingredients

- **Compiler** – split original code into host and device parts and compile them into single object
  - Code splitter (source-to-source preprocessor)
  - Target device code generator
- **Runtime library** – implementation of specific internal functions used in generated code
  - Data management
  - Kernel invocation
  - Kernel results verification

Priorities

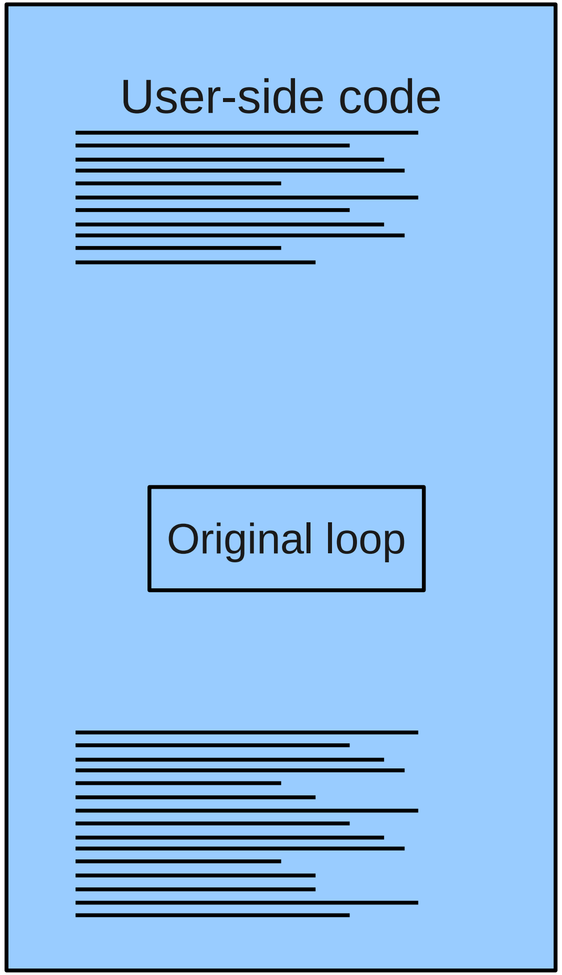
1) Make up the rough version of the full toolchain first, focus on improvements later

2) Use empirical tests where analysis is not yet sufficient (e.g. for identifying parallel loops)

3) Focus on best compiled kernels yield (code coverage) for COSMO and other models

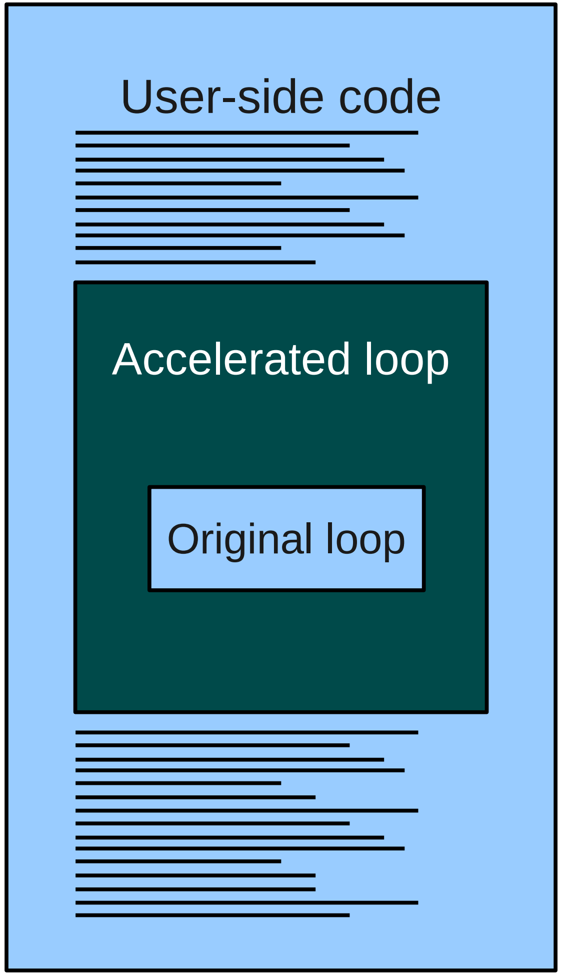
4) Implement optimizations later

Runtime workflow



We start with original source code, selecting loops suitable for device acceleration.

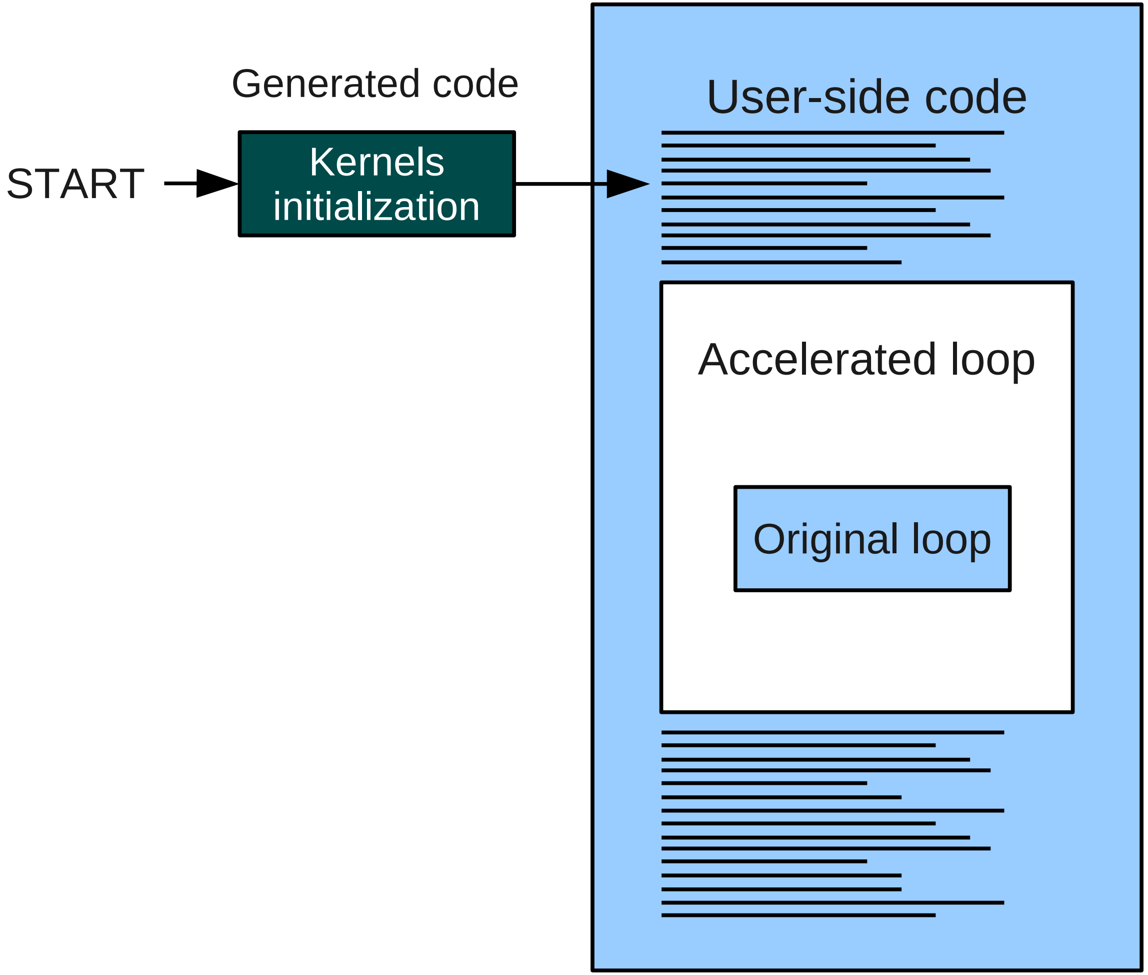
Runtime workflow



Equivalent device code is generated for suitable loops.

(see “Code generation workflow” for details)

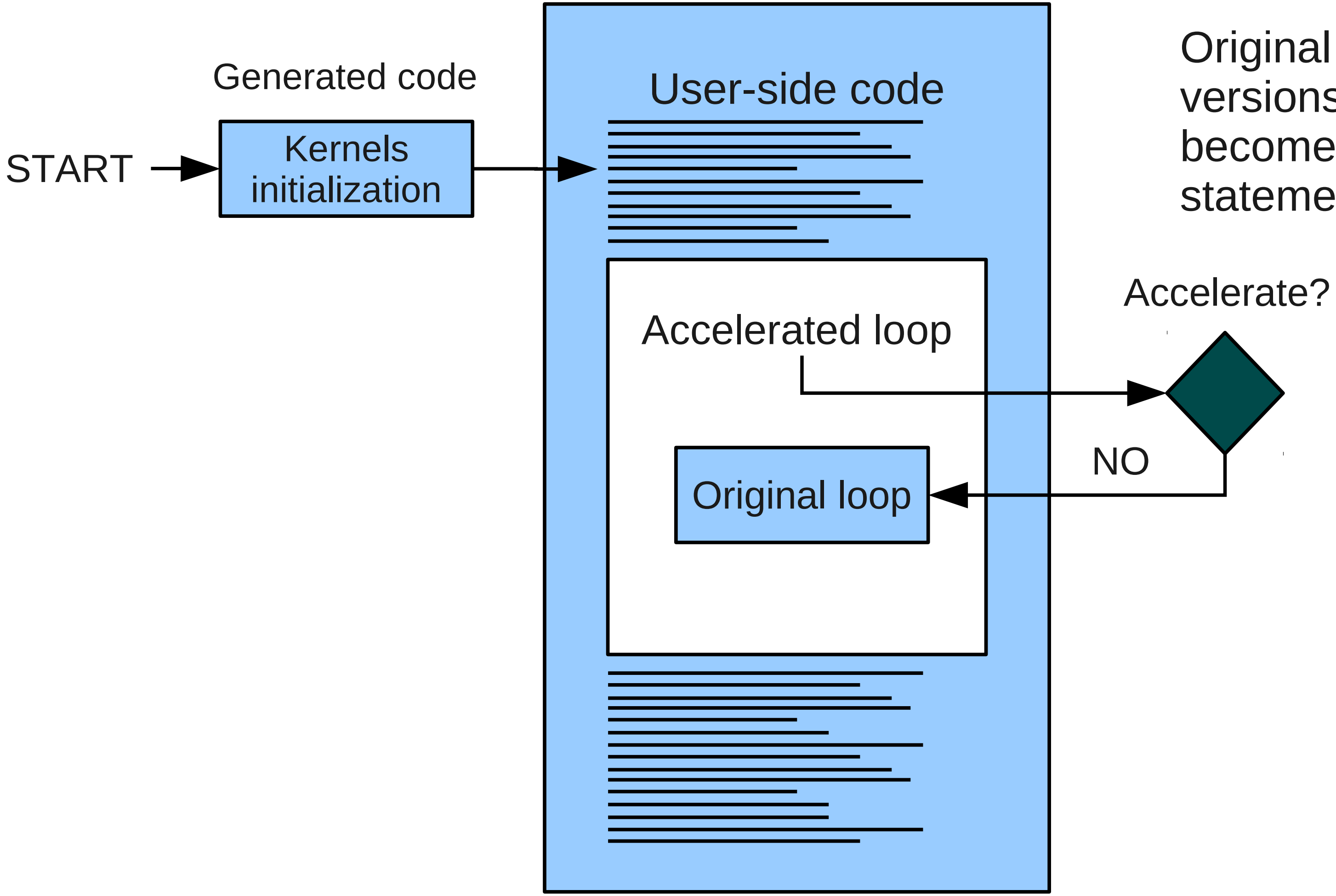
Runtime workflow



Equivalent device code is generated for suitable loops.

Additionally global constructors are generated to initialize configuration structures (with status, profiling, permanent dependencies, etc.) for each kernel.

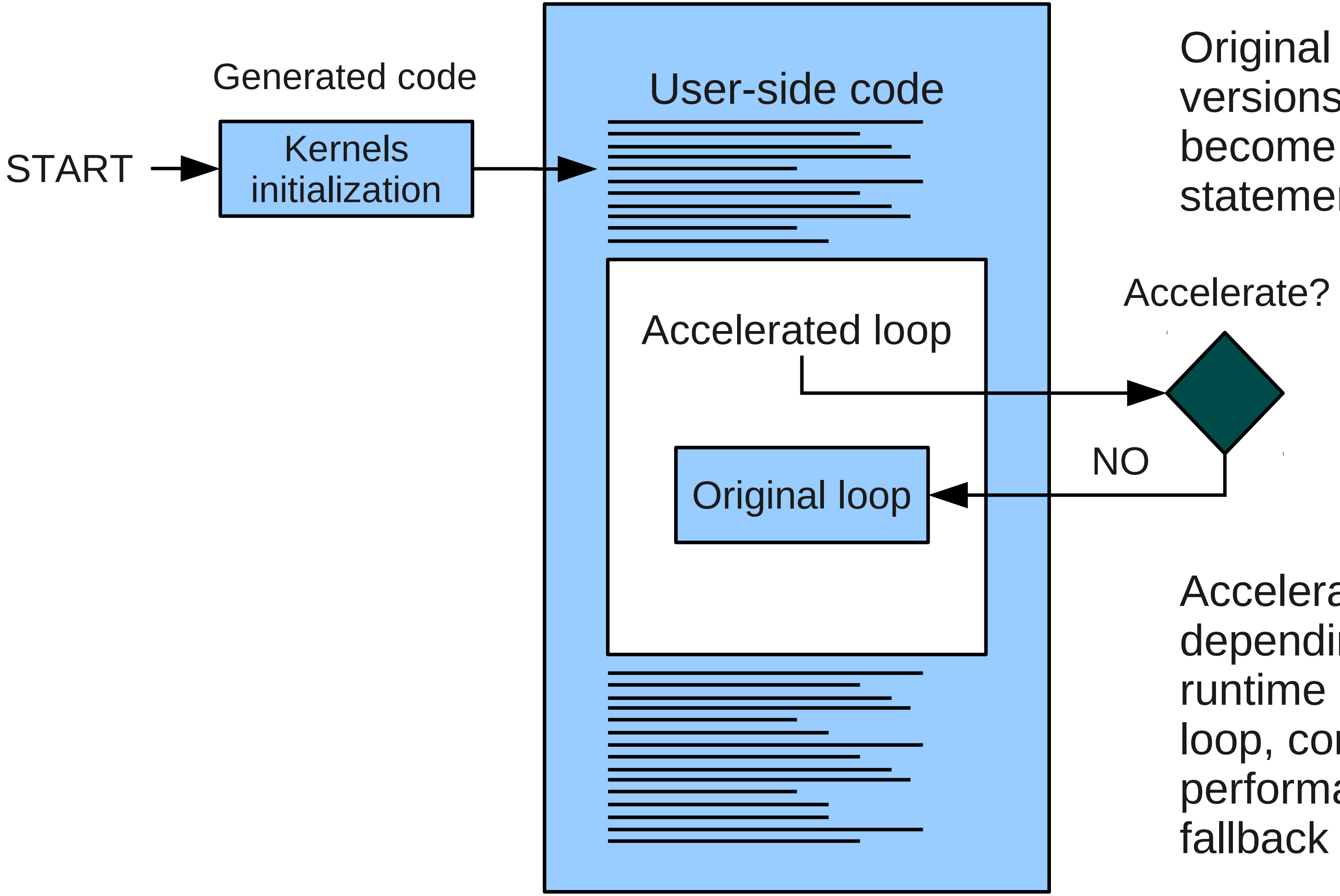
Runtime workflow



Original and accelerated versions of the same loop become two branches of if-statement.

2011

Runtime workflow

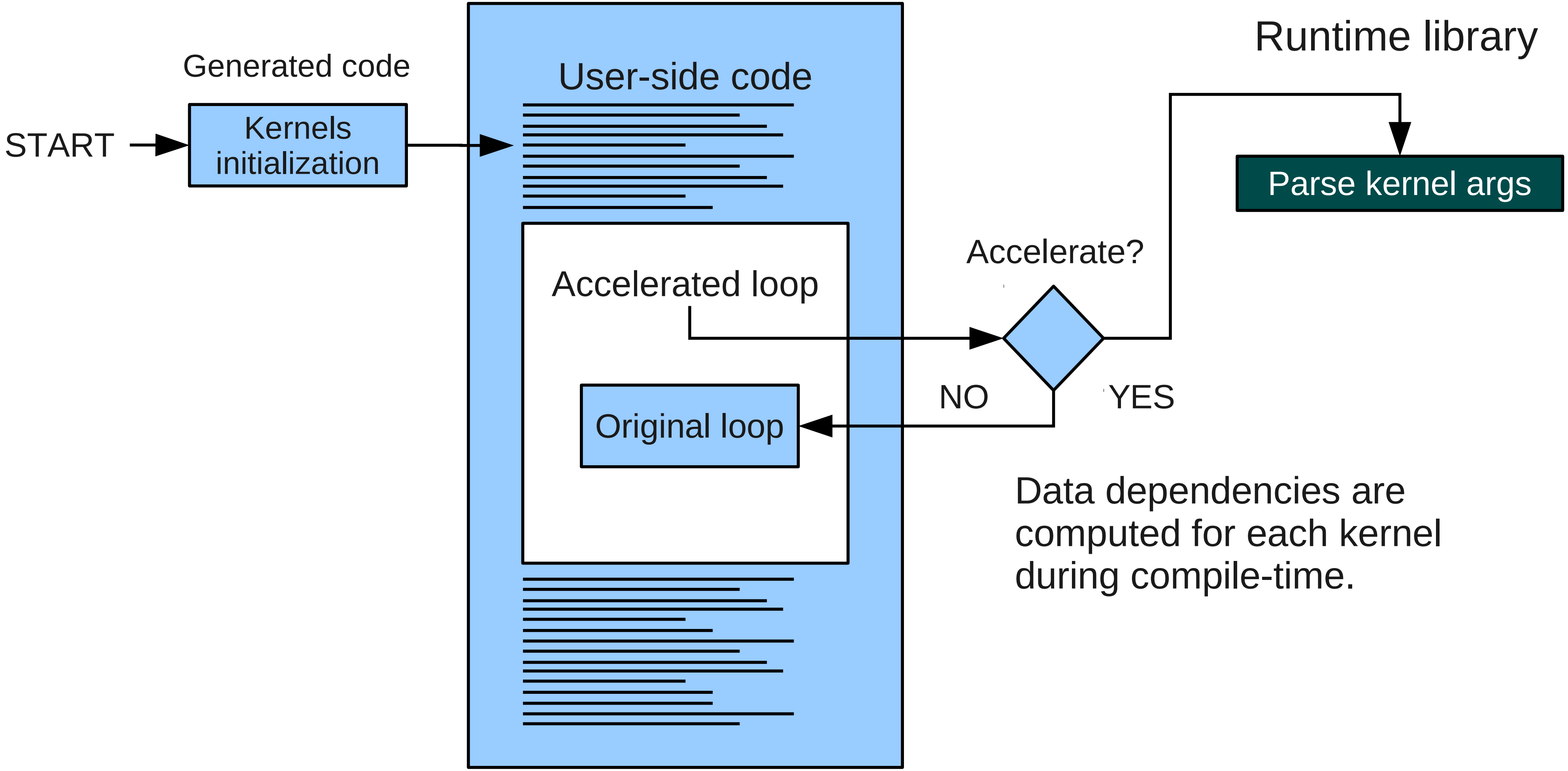


Original and accelerated versions of the same loop become two branches of if-statement.

Accelerated loop is selected, depending on the number of runtime properties (not failing loop, correct results, better performance, etc.); otherwise – fallback to original version.

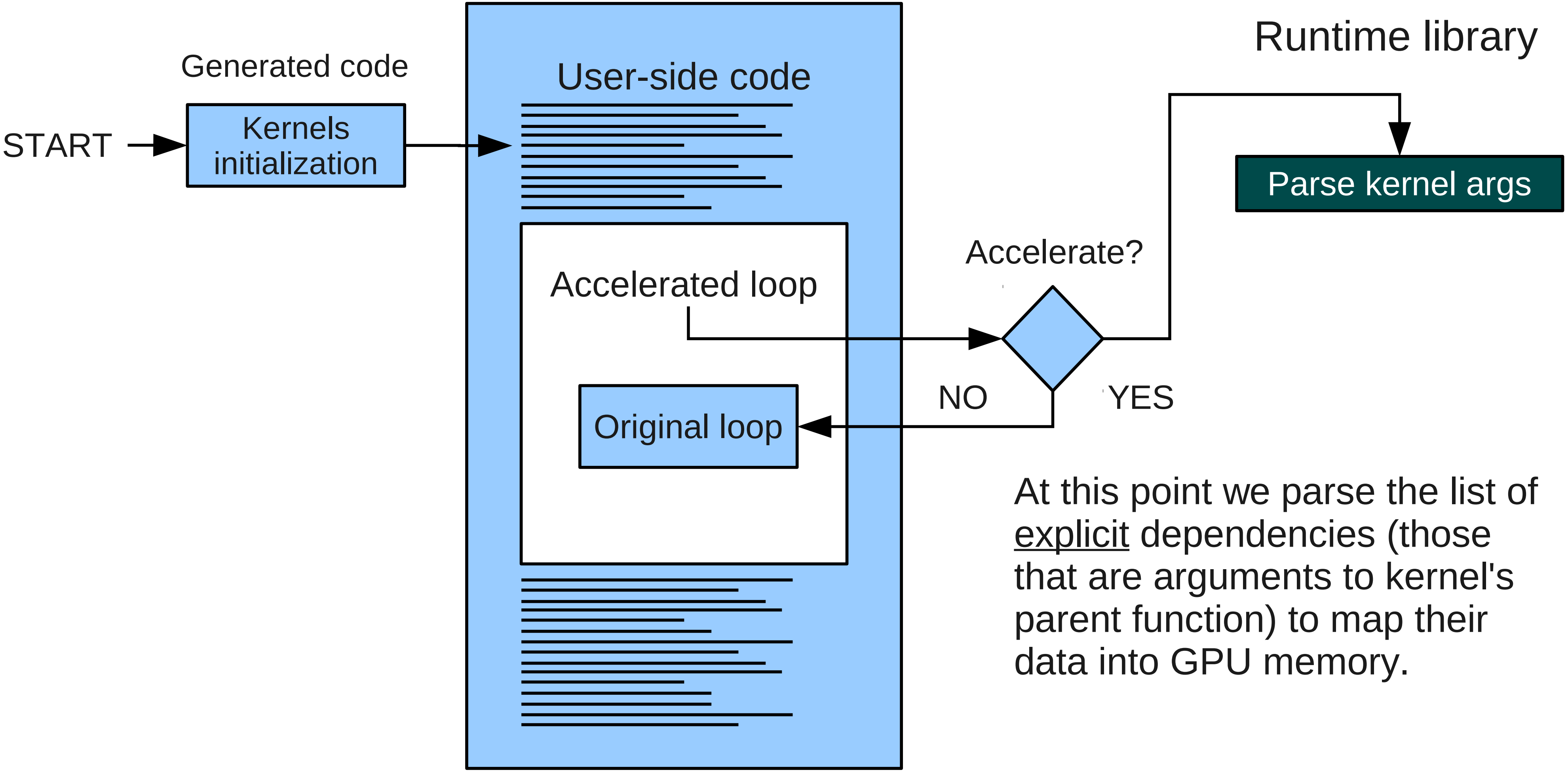
2011

Runtime workflow



Data dependencies are computed for each kernel during compile-time.

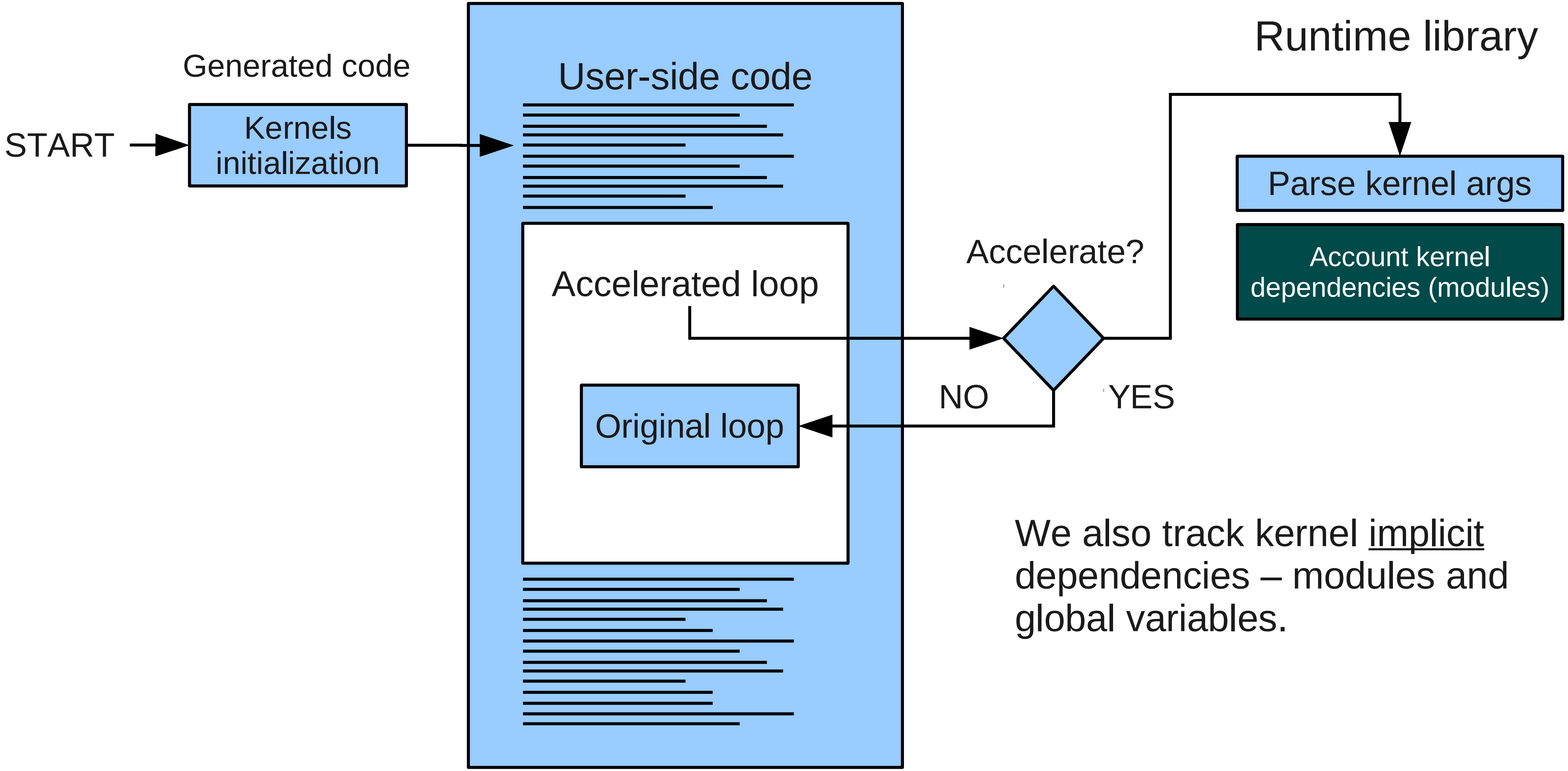
Runtime workflow



At this point we parse the list of <u>explicit</u> dependencies (those that are arguments to kernel's parent function) to map their data into GPU memory.

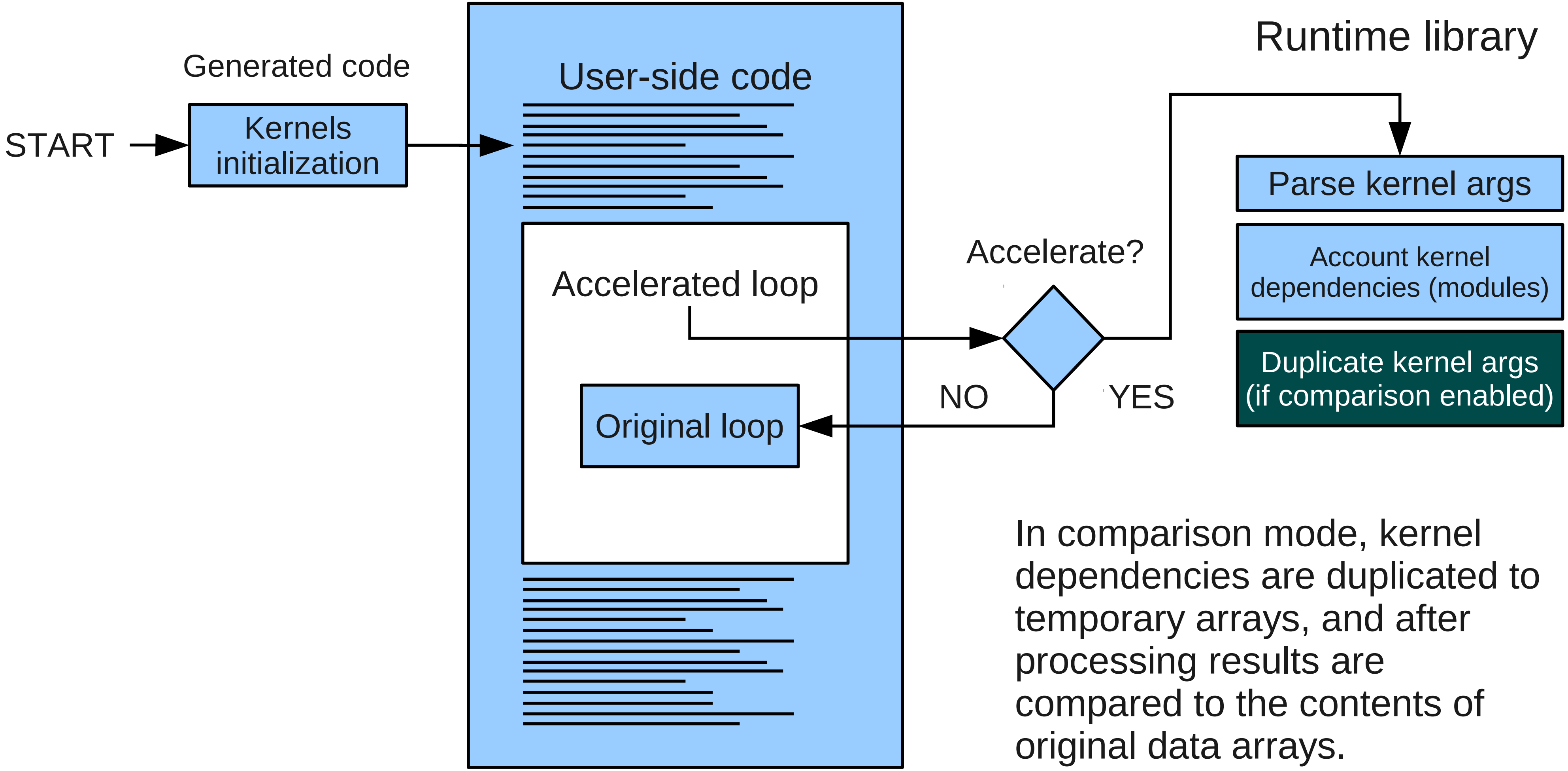
2011

Runtime workflow



We also track kernel implicit dependencies – modules and global variables.

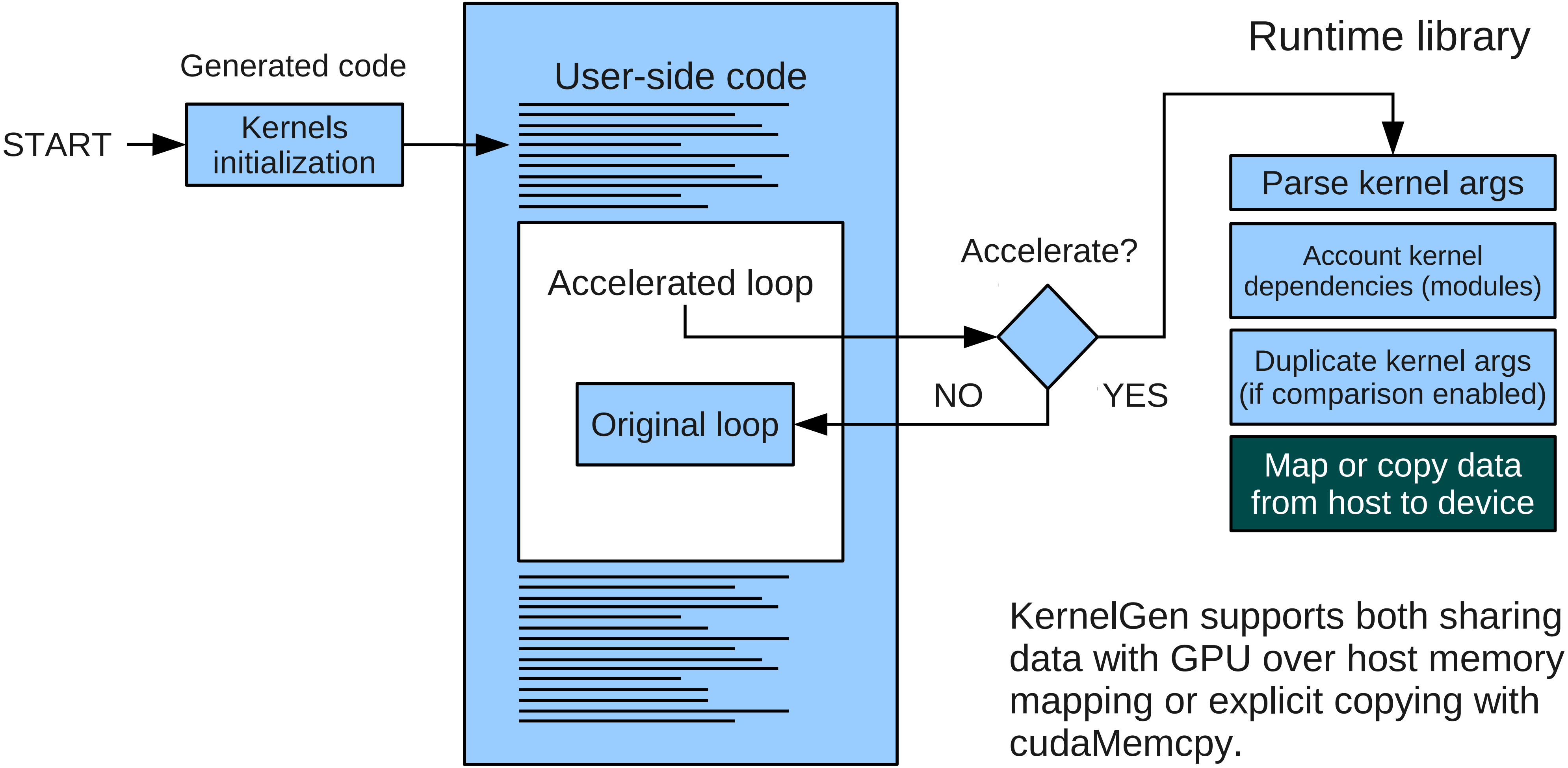
Runtime workflow



In comparison mode, kernel dependencies are duplicated to temporary arrays, and after processing results are compared to the contents of original data arrays.

2011

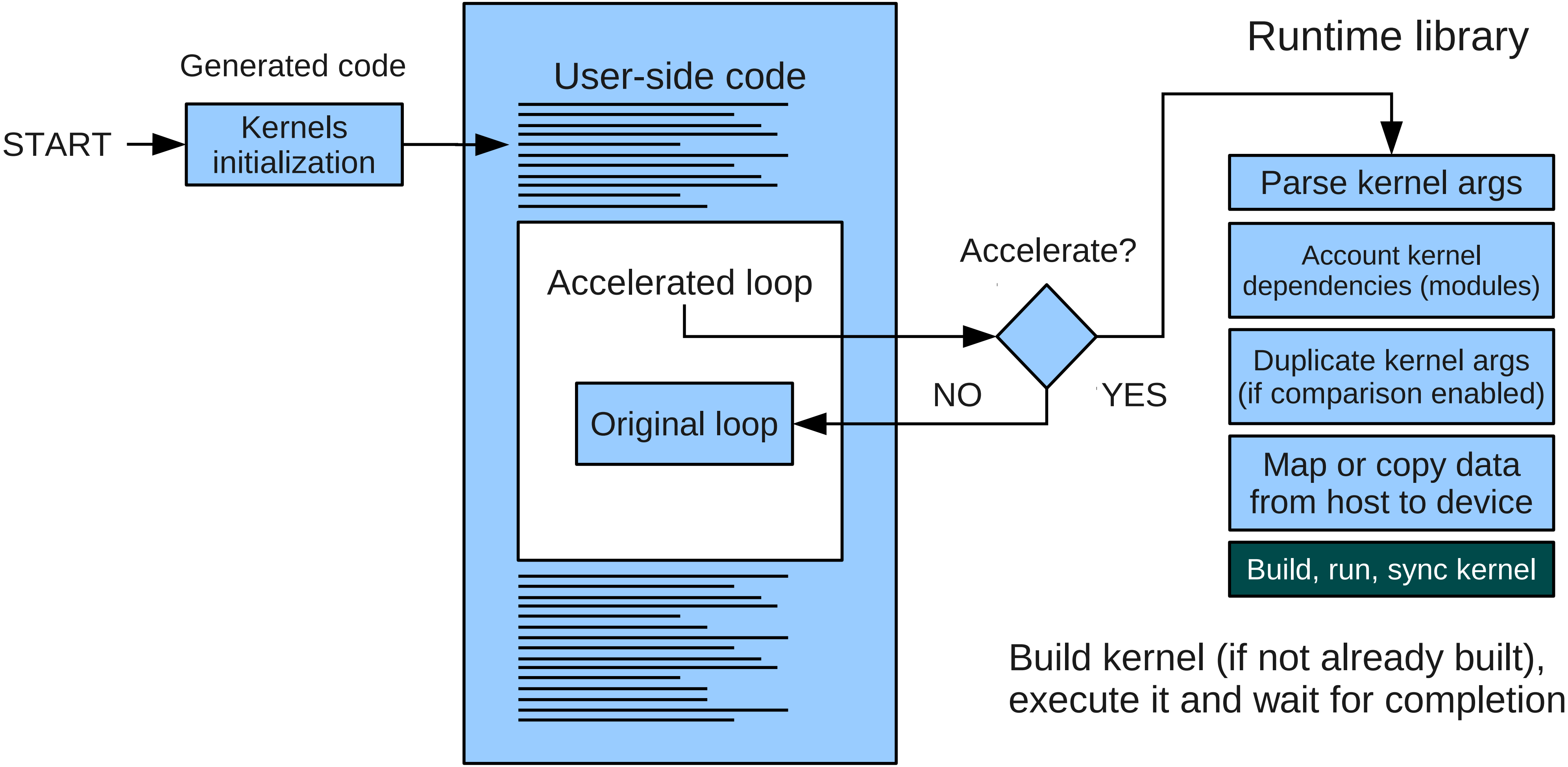
Runtime workflow



KernelGen supports both sharing data with GPU over host memory mapping or explicit copying with cudaMemcpy.

2011

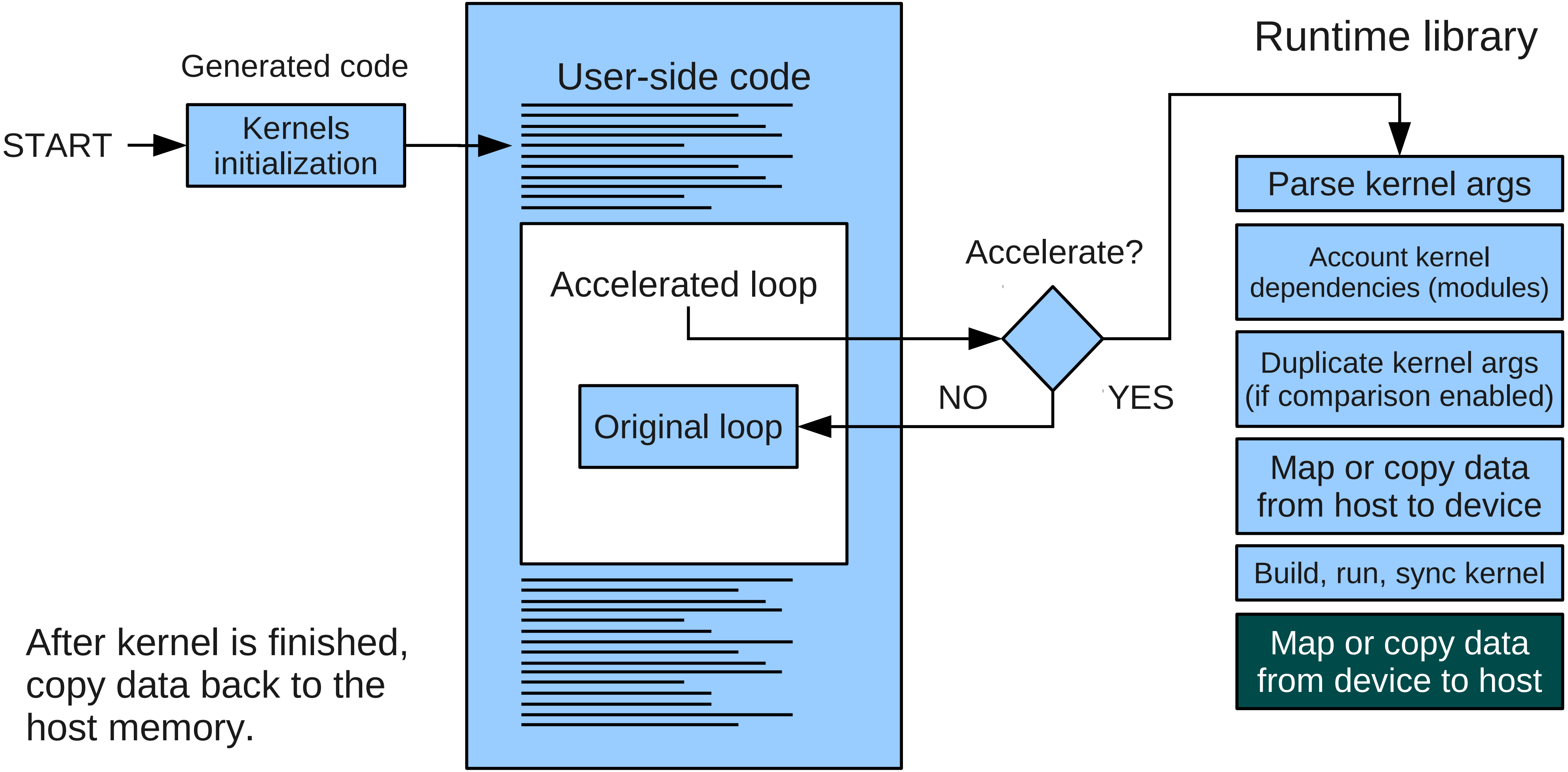
Runtime workflow



Build kernel (if not already built), execute it and wait for completion.

2011

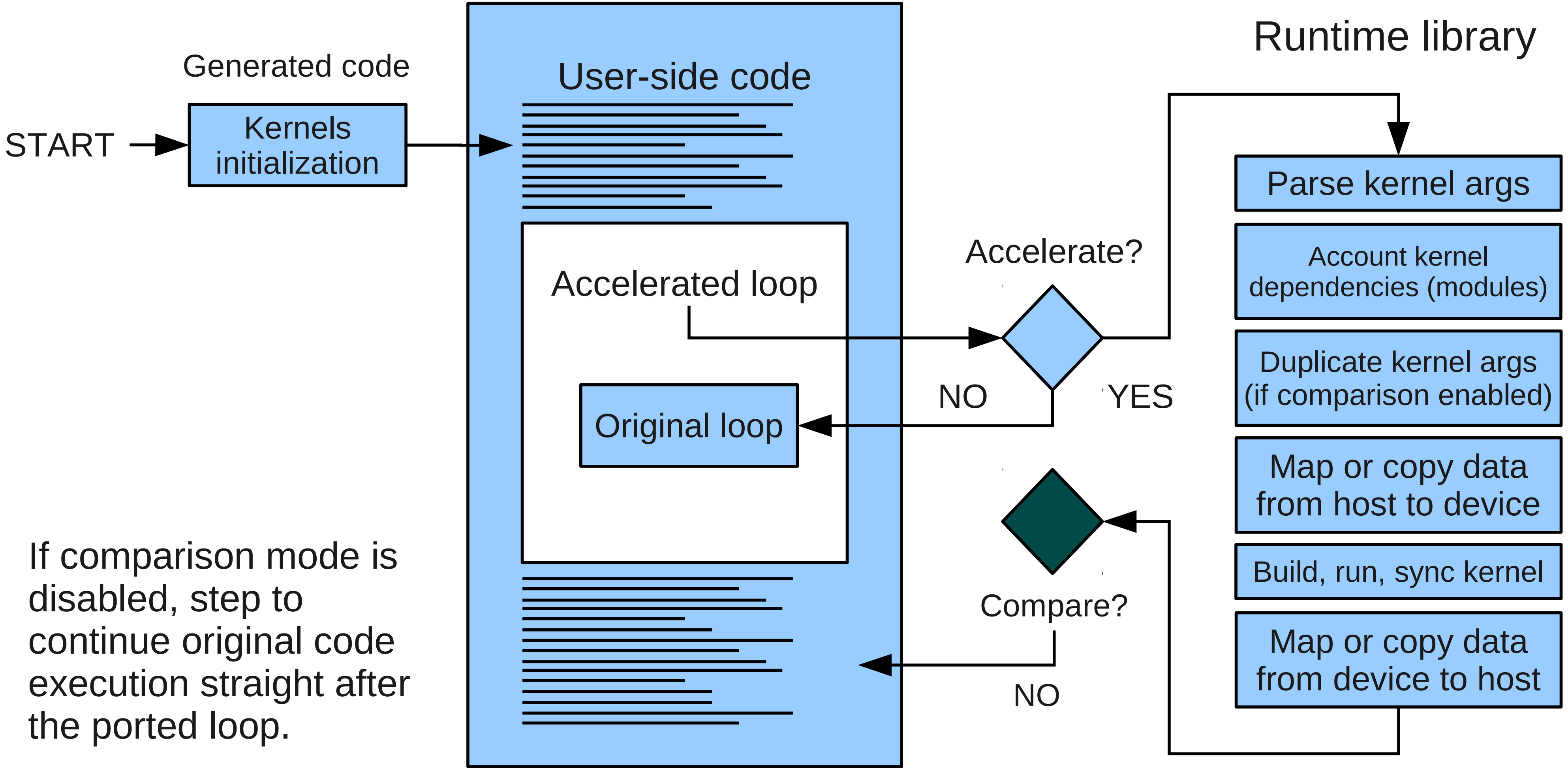
Runtime workflow



After kernel is finished, copy data back to the host memory.

2011

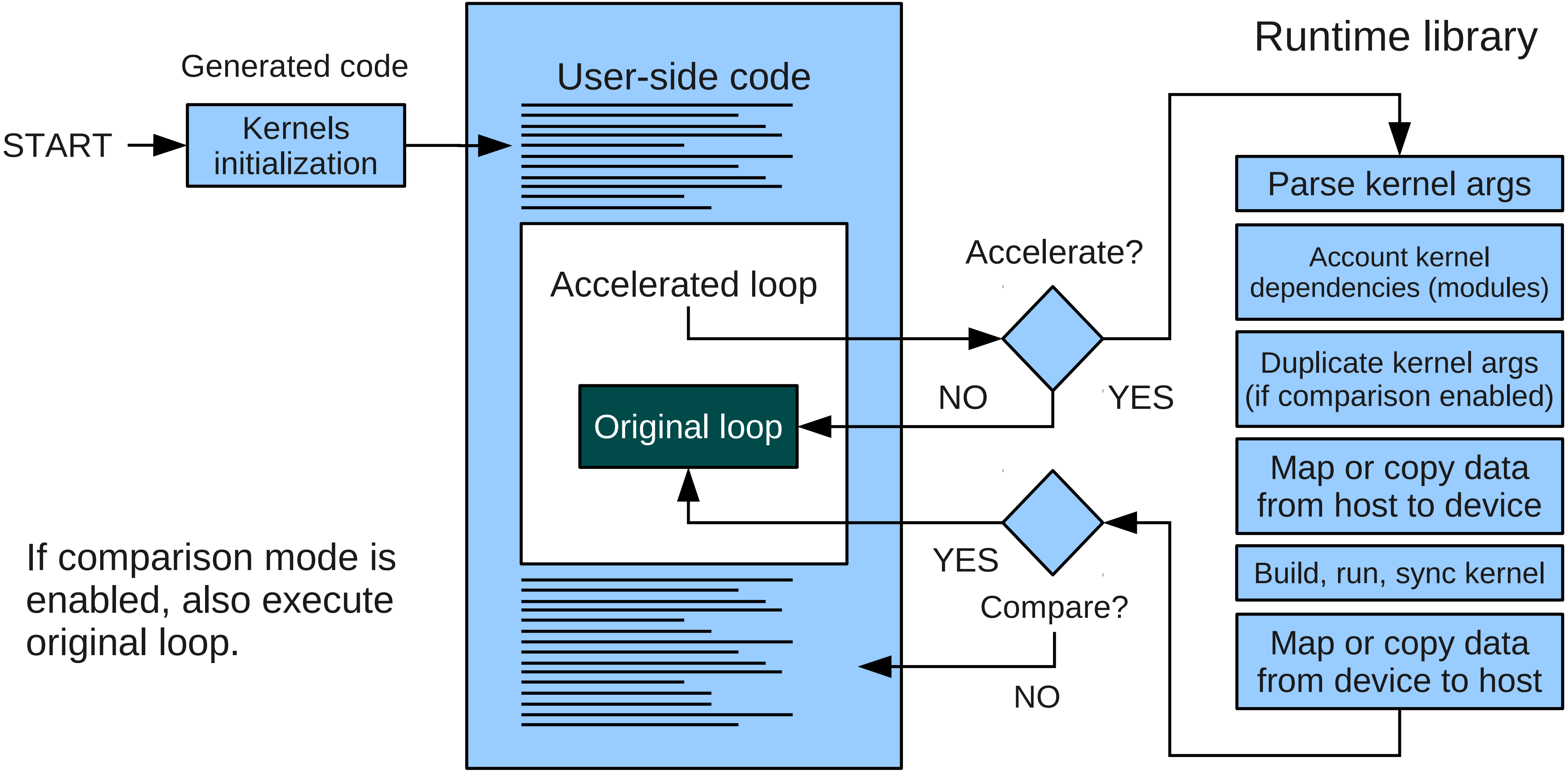
Runtime workflow



If comparison mode is disabled, step to continue original code execution straight after the ported loop.

2011

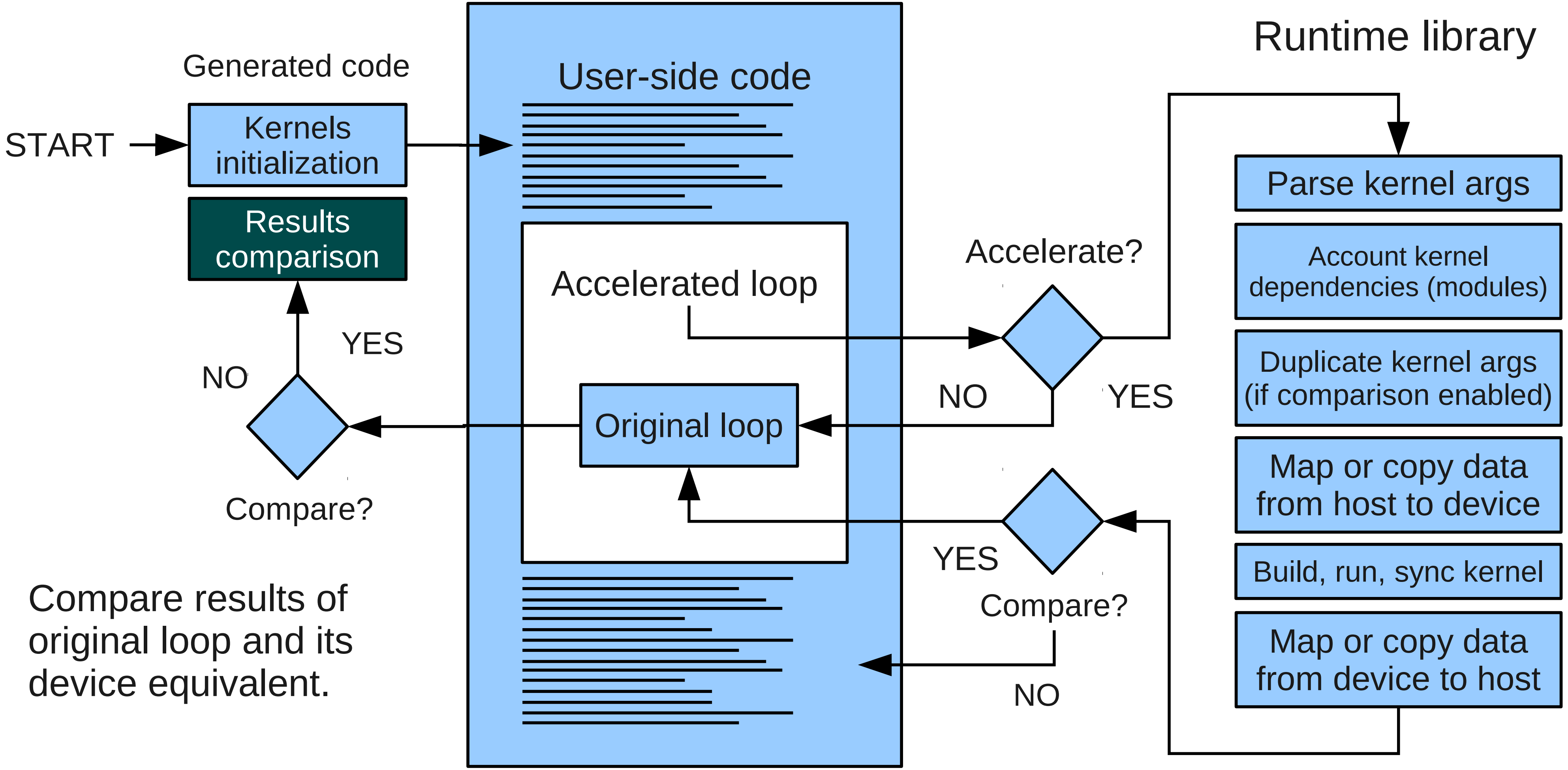
Runtime workflow



If comparison mode is enabled, also execute original loop.

2011

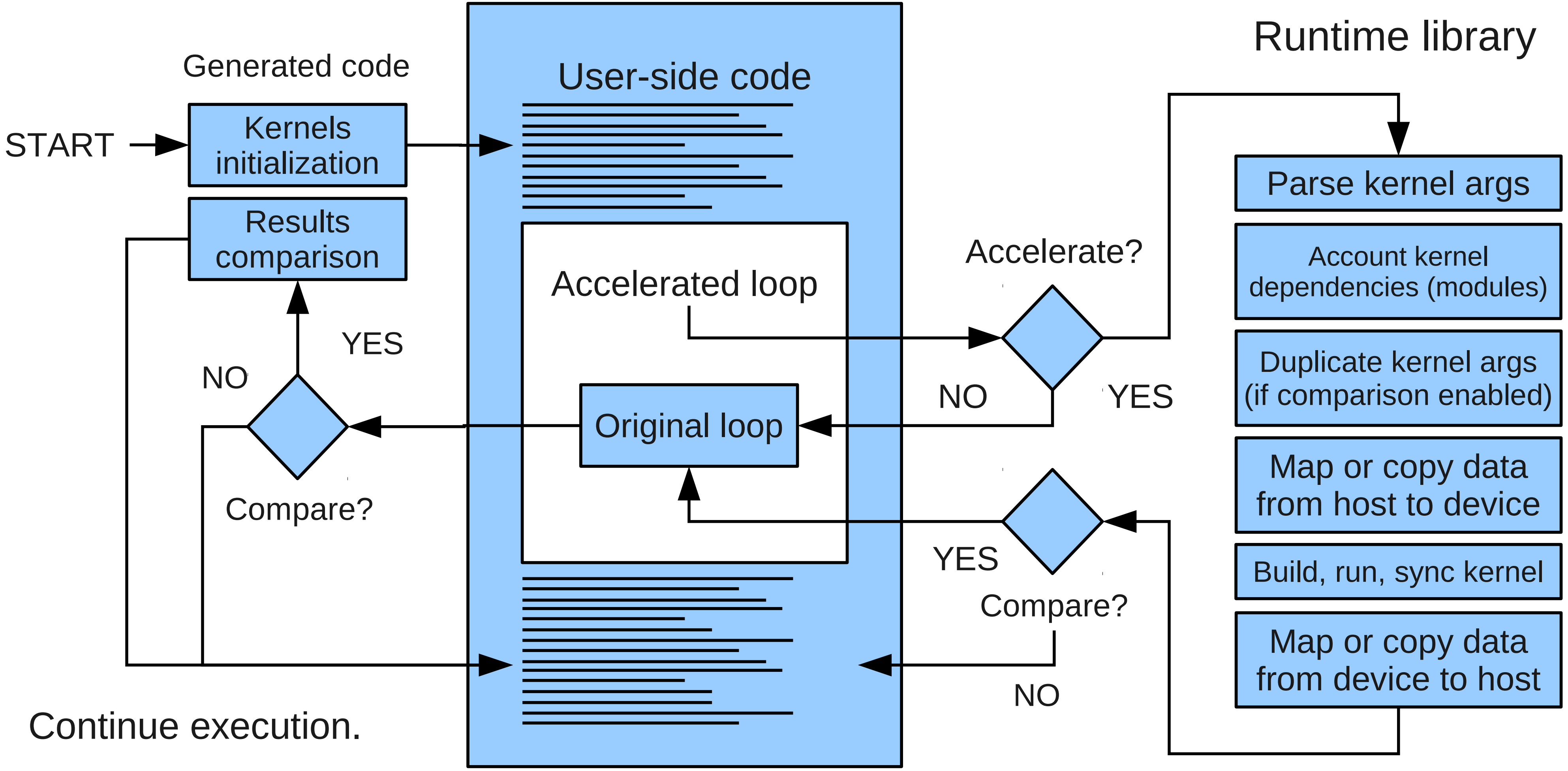
Runtime workflow



Compare results of original loop and its device equivalent.

2011

Runtime workflow



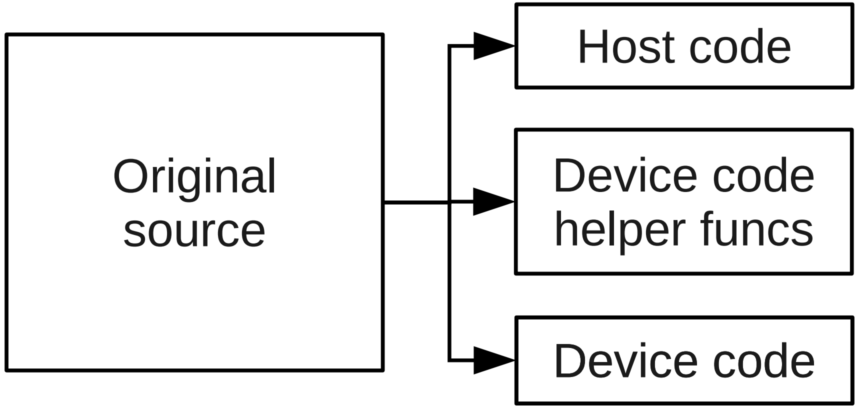
2011

Code generation workflow

Two parts of code generation process:

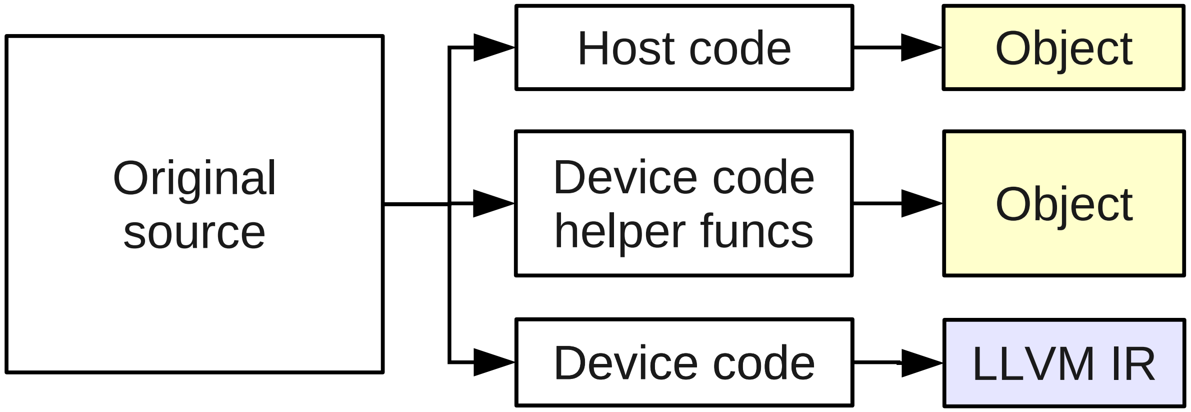
- **Compile time** – generate kernels strictly corresponding to original host loops
- **Runtime** – generate kernels, using additional info available at runtime: inline external functions, optimize compute grid, etc.

Code generation workflow (compile-time part)



Loops suitable for device execution are identified in original source code, their bodies are surrounded with if-statement to switch between original loop and call to device kernel for this loop. Each suitable loop is duplicated in form of subroutine in a separate compilation unit. Additionally, helper initialization anchors are generated.

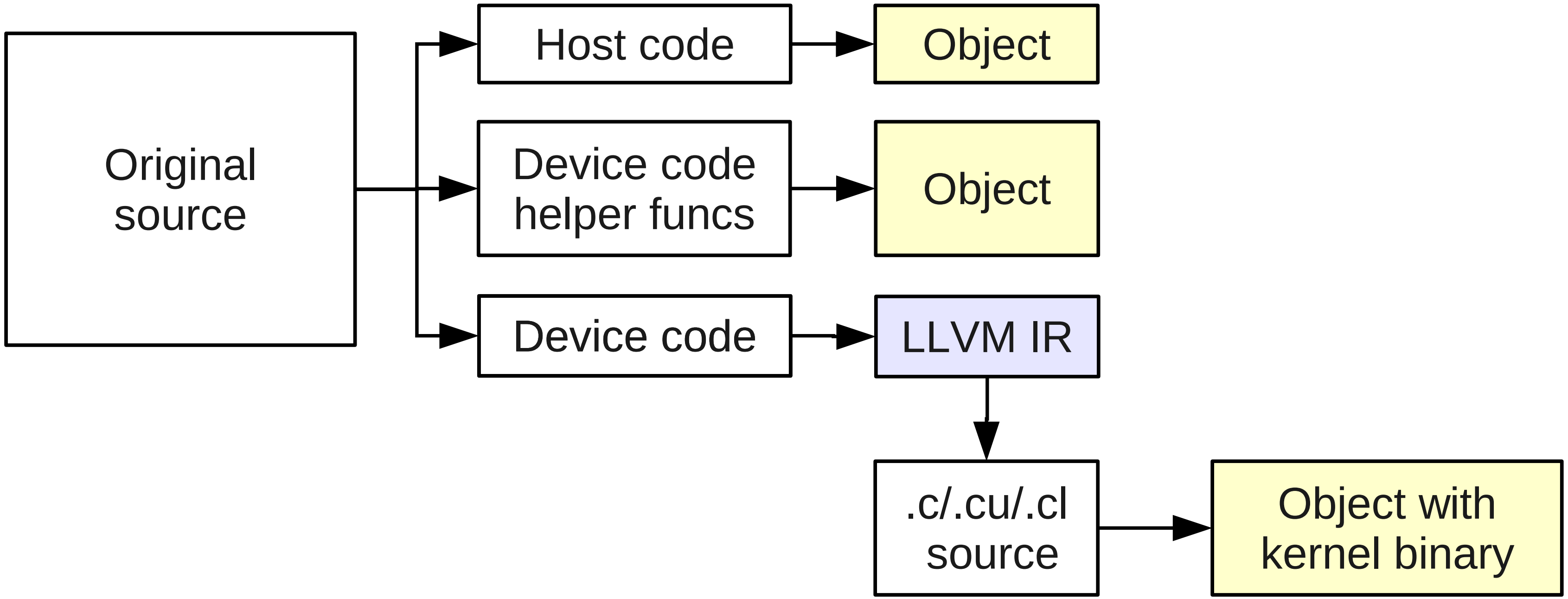
Code generation workflow (compile-time part)



Objects for host code and device code helper functions can be generated directly with CPU compiler used by application.

Device code is compiled into Low-Level Virtual Machine Intermediate representation (LLVM IR).

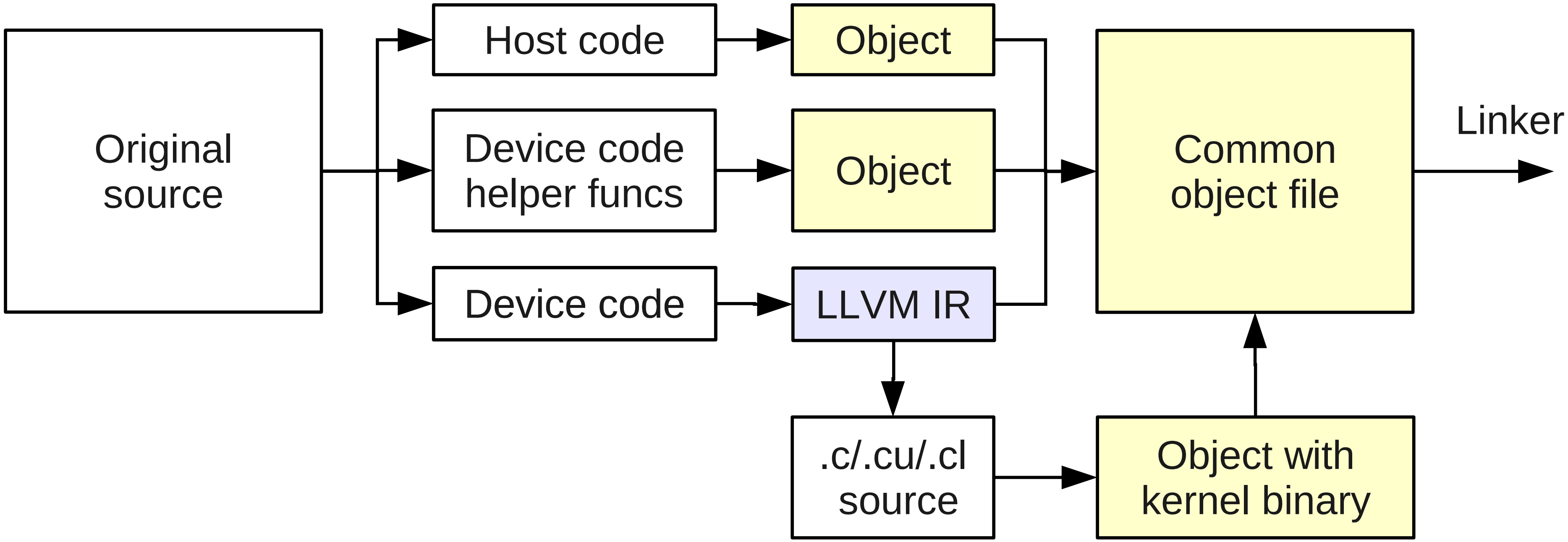
Code generation workflow (compile-time part)



Code from LLVM IR is translated into C, CUDA or OpenCL using modified LLVM C Backend and compiled using the corresponding device compiler.

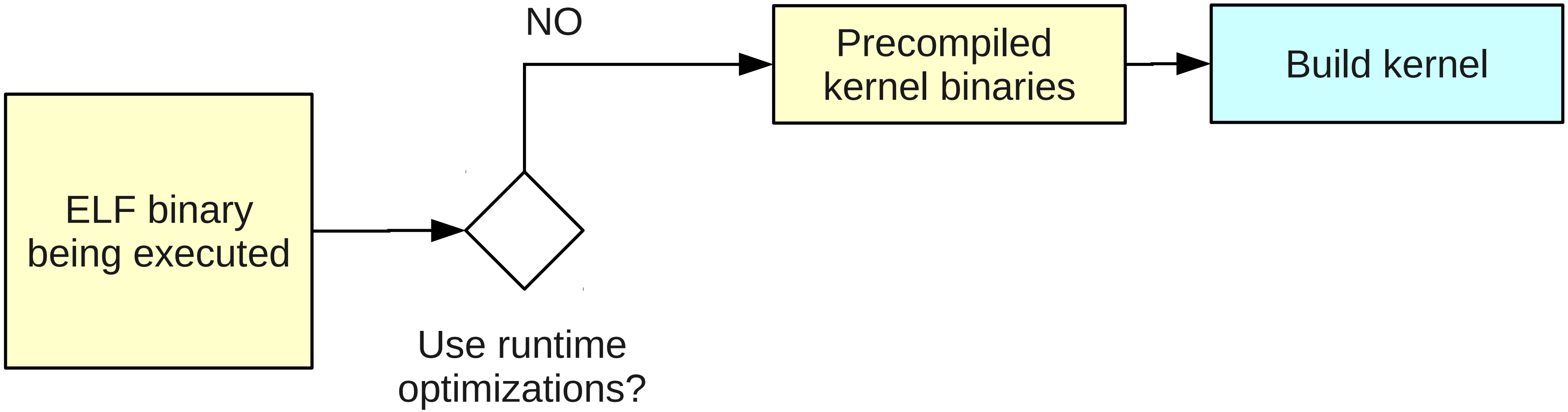
2011

Code generation workflow (compile-time part)



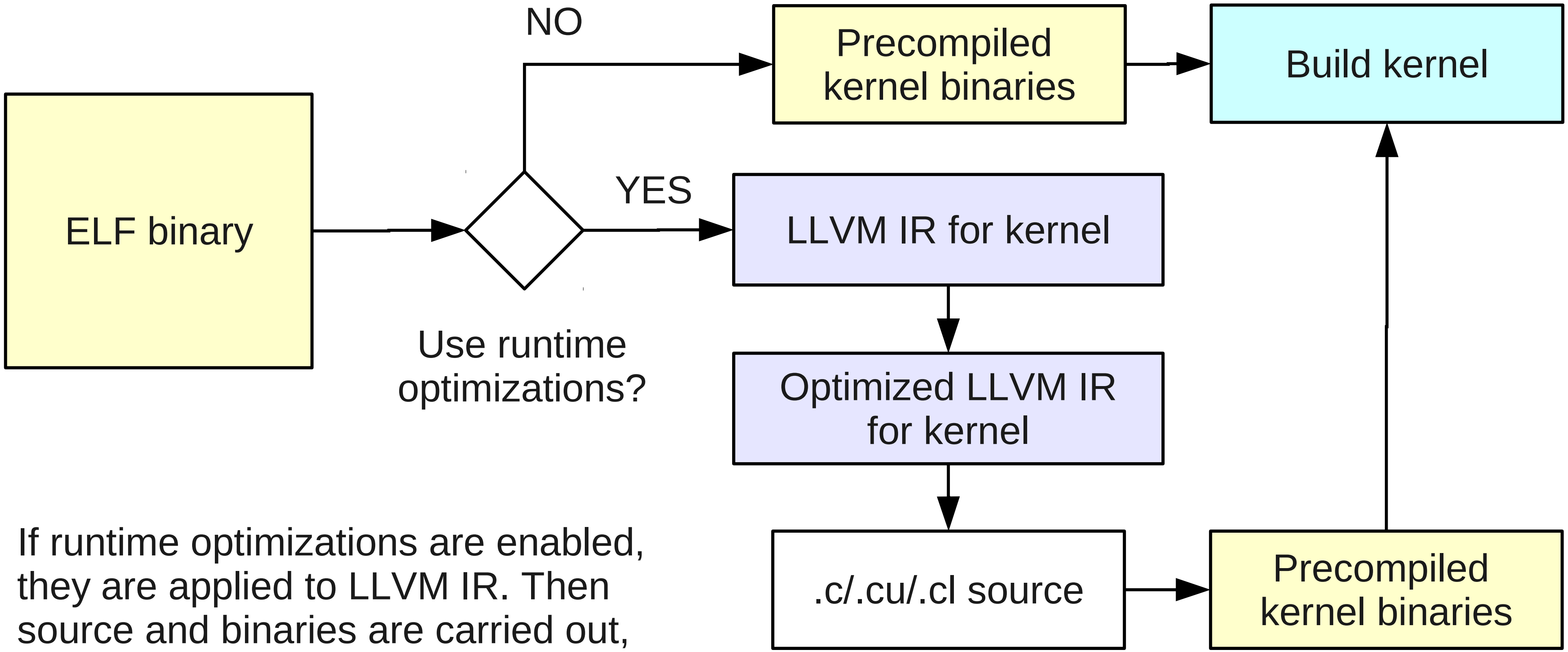
Finally, objects for all parts of the code are merged into single object to conserve "1 source → 1 object" layout. LLVM IR is also embedded into resulting object.

Code generation workflow (runtime part)


NO
Precompiled kernel binaries
Build kernel
ELF binary being executed
Use runtime optimizations?

Without runtime optimizations enabled, the previously compiled kernel binary could be built and executed.

Code generation workflow (runtime part)



NO
Precompiled kernel binaries
Build kernel
ELF binary
YES
LLVM IR for kernel
Use runtime optimizations?
Optimized LLVM IR for kernel
.c/.cu/.cl source
Precompiled kernel binaries

If runtime optimizations are enabled, they are applied to LLVM IR. Then source and binaries are carried out, just like in compile-time process.

3. Toolchain internals

2011

Example: sincos

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

```fortran
subroutine sincos(nx, ny, nz, x, y, xy)

implicit none

integer, intent(in) :: nx, ny, nz
real, intent(in) :: x(nx, ny, nz), y(nx, ny, nz)
real, intent(inout) :: xy(nx, ny, nz)

integer :: i, j, k

do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo

end subroutine sincos
```

1: host part of code split (1/3)

```
module sincos_kernelgen_module_uses
end module sincos_kernelgen_module_uses
module sincos_kernelgen_module
USE KERNELGEN

type(kernelgen_kernel_config), bind(C) :: sincos_loop_1_kernelgen_config

interface
function sincos_loop_1_kernelgen_compare()
end function

end interface

end module sincos_kernelgen_module


subroutine sincos(nx, ny, nz, x, y, xy)

USE KERNELGEN
USE sincos_kernelgen_module


implicit none
```

1: host part of code split (1/3)

```
module sincos_kernelgen_module_uses
end module sincos_kernelgen_module_uses
module sincos_kernelgen_module
USE KERNELGEN

type(kernelgen_kernel_config), bind(C) :: sincos_loop_1_kernelgen_config

interface
function sincos_loop_1_kernelgen_compare()
end function

end interface

end module sincos_kernelgen_module


subroutine sincos(nx, ny, nz, x, y, xy)

USE KERNELGEN
USE sincos_kernelgen_module


implicit none
```

Per-kernel config structure

1: host part of code split (1/3)

```
module sincos_kernelgen_module_uses
end module sincos_kernelgen_module_uses
module sincos_kernelgen_module
USE KERNELGEN

type(kernelgen_kernel_config), bind(C) :: sincos_loop_1_kernelgen_config

interface
function sincos_loop_1_kernelgen_compare()
end function

end interface

end module sincos_kernelgen_module


subroutine sincos(nx, ny, nz, x, y, xy)

USE KERNELGEN
USE sincos_kernelgen_module


implicit none
```

Adding kernel-specific and internal module with runtime calls

1: host part of code split (2/3)

```fortran
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$KERNELGEN CALL sincos_loop_1_kernelgen
  call kernelgen_launch(sincos_loop_1_kernelgen_config, 1, nx, 1, ny, 1, nz, 6, 0,
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERNELGEN LOOP sincos_loop_1_kernelgen
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

1: host part of code split (2/3)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_
!$KERNELGEN CALL s
  call kernelgen_l                                                 0,
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERNELGEN LOOP sincos_loop_1_kernelgen
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

Loop location marker for processing script to clear everything here, if kernel was not successfully compiled.

1: host part of code split (2/3)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$KERNELGEN CALL sinco
  call kernelgen_launc
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERNELGEN LOOP sincos_loop_1_kernelgen
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

If kernel is requested to be executed not only on host

1: host part of code split (2/3)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$KERNELGEN CALL sincos_loop_1_kernelgen
  call kernelgen_launch(sincos_loop_1_kernelgen_config, 1, nx, 1, ny, 1, nz, 6, 0,
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERNELGEN LOOP sincos_loop_1_kernelgen
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

Launch kernel with its config handle, grid and dependencies

1: host part of code split (2/3)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$KERNELGEN CALL sincos_loop_1_kernelgen
  call kernelgen_launch(sincos_loop_1_kernelgen_config, 1, nx, 1, ny, 1, nz, 6, 0,
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNE
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERNELGEN LOOP sincos_loop_1_kernelgen
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

Just in case increment old indexes, like if they were used by loop

1: host part of code split (2/3)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$KERNELGEN CALL sincos_loop_1_kernelgen
  call kernelgen_launch(sincos_loop_1_kernelgen_config, 1, nx, 1, ny, 1, nz, 6, 0,
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERNE
do k =
  do j
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

If kernel is requested to be executed not only on host
or there is an error executing kernel on device

1: host part of code split (2/3)

Execute original loop

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$KERNELGEN CALL sincos_loop_1_kernelgen
  call kernelgen_launch(sincos_loop_1_kernelgen_config, 1, nx, 1, ny, 1, nz, 6, 0,
nz, sizeof(nz), nz, ny, sizeof(ny), ny, nx, sizeof(nx), nx, xy, sizeof(xy), xy, x,
sizeof(x), x, y, sizeof(y), y)
k = nz + 1
j = ny + 1
i = nx + 1
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (k                                e. 0)) then
!$KERNE                                elgen
do k = 1, nz
  do j = 1, ny
    do i = 1, nx
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
    enddo
  enddo
enddo
!$KERNELGEN END LOOP sincos_loop_1_kernelgen
endif
```

1: host part of code split (3/3)

```
if ((sincos_loop_1_kernelgen_config%compare .eq. 1) .and. (kernelgen_get_last_error()
.eq. 0)) then
  call kernelgen_compare(sincos_loop_1_kernelgen_config,
sincos_loop_1_kernelgen_compare, kernelgen_compare_maxdiff)
endif
!$KERNE

end subroutine sincos
```

If no error and comparison enabled, compare results of CPU and device

2: device part of code split (1/2)

```
subroutine sincos_loop_1_kernelgen(nz, ny, nx, xy, x, y)
implici
interfa
subroutine sincos_loop_1_kernelgen_blockidx_x(index, start, end) bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
subroutine sincos_loop_1_kernelgen_blockidx_y(index, start, end) bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
subroutine sincos_loop_1_kernelgen_blockidx_z(index, start, end) bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
end interface
```

Kernel subroutine name is a decorated name of original loop function

2: device part of code split (1/2)

```
subroutine sincos_loop_1_kernelgen(nz, ny, nx, xy, x, y)
implicit none
interface
subroutine sincos_loop_1_kernelgen_blockidx_x(index, start, end) bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
subroutine sincos_loop_1_kernelgen_blockidx_y(index, start, end) bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
subroutine sincos_loop_1_kernelgen_blockidx_z(index, start, end) bind(C)
use iso_c_binding
integer(c_int) :: index
integer(c_int), value :: start, end
end subroutine
end interface
```

Interfaces to device functions returning device compute grid dimensions

2: device part of code split (2/2)

```
#ifdef __CUDA_DEVICE_FUNC__
call sincos_loop_1_kernelgen_blockidx_z(k, 1, nz)
#else
do k = 1
#endif
#ifdef __CUDA_DEVICE_FUNC__
call sincos_loop_1_kernelgen_blockidx_y(j, 1, ny)
#else
do j = 1, ny
#endif
#ifdef __CUDA_DEVICE_FUNC__
call sincos_loop_1_kernelgen_blockidx_x(i, 1, nx)
#else
do i = 1, nx
#endif
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
#ifndef __CUDA_DEVICE_FUNC__
enddo
#endif
#ifndef __CUDA_DEVICE_FUNC__
enddo
#endif
#ifndef __CUDA_DEVICE_FUNC__
enddo
#endif
end subroutine sincos_loop_1_kernelgen
```

In device kernels loops indexes are computed using block/thread indexes

2: device part of code split (2/2)

```
#ifdef __CUDA_DEVICE_FUNC__
call sincos_loop_1_kernelgen_blockidx_z(k, 1, nz)
#else
do k = 1, nz
#endif
#ifdef __CUDA_DEVICE_FUNC__
call sincos_loop_1_kernelgen_blockidx_y(j, 1, ny)
#else
do j = 1, ny
#endif
#ifdef __CUDA_DEVICE_FUNC__
call sincos_loop_1_kernelgen_blockidx_x(i, 1, nx)
#else
do i = 1, nx
#endif
      xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))

#ifndef
enddo
#endif
#ifndef __CUDA_DEVICE_FUNC__
enddo
#endif
#ifndef __CUDA_DEVICE_FUNC__
enddo
#endif
end subroutine sincos_loop_1_kernelgen
```

The body of original loop

3: LLVM IR for device code (1/2)

```
; ModuleID = 'sincos.sincos_loop_1_kernelgen.cuda.device.F90.ir'
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 20110810 (prerelease) LLVM: 136347M\22"

define void @sincos_loop_1_kernelgen_(i32* nocapture %nz, i32* nocapture %ny, i32* nocapture %nx, [0
x float]* %xy, [0 x float]* %x, [0 x float]* %y) nounwind uwtable {
entry:
  %memtmp = alloca i32, align 4
  %memtmp3 = alloca i32, align 4
  %memtmp4 = alloca i32, align 4
  %0 = load i32* %nx, align 4
  %1 = sext i32 %0 to i64
  %2 = icmp slt i64 %1, 0
  %3 = select i1 %2, i64 0, i64 %1
  %4 = load i32* %ny, align 4
  %5 = sext i32 %4 to i64
  %6 = mul nsw i64 %3, %5
  %7 = icmp slt i64 %6, 0
  %8 = select i1 %7, i64 0, i64 %6
  %not = xor i64 %3, -1
  %9 = sub nsw i64 %not, %8
  %10 = load i32* %nz, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_z(i32* noalias %memtmp, i32 1,
i32 %10) nounwind
  %11 = load i32* %ny, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_y(i32* noalias %memtmp3, i32 1,
i32 %11) nounwind
```

2011

3: LLVM IR for device code (1/2)

```
; ModuleID = 'sincos.sincos_loop_1_kernelgen.cuda.device.F90.ir'
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 20110810 (prerelease) LLVM: 136347M\22"

define void @sincos_loop_1_kernelgen_(i32* nocapture %nz, i32* nocapture %ny, i32* nocapture %nx, [0
x float]* %xy, [0 x float]* %x, [0 x float]* %y) nounwind uwtable {
entry:
  %memtmp
  %memtmp3 = alloca i32, align 4
  %memtmp4 = alloca i32, align 4
  %0 = load i32* %nx, align 4
  %1 = sext i32 %0 to i64
  %2 = icmp slt i64 %1, 0
  %3 = select i1 %2, i64 0, i64 %1
  %4 = load i32* %ny, align 4
  %5 = sext i32 %4 to i64
  %6 = mul nsw i64 %3, %5
  %7 = icmp slt i64 %6, 0
  %8 = select i1 %7, i64 0, i64 %6
  %not = xor i64 %3, -1
  %9 = sub nsw i64 %not, %8
  %10 = load i32* %nz, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_z(i32* noalias %memtmp, i32 1,
i32 %10) nounwind
  %11 = load i32* %ny, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_y(i32* noalias %memtmp3, i32 1,
i32 %11) nounwind
```

```
subroutine sincos_loop_1_kernelgen(nz, ny, nx, xy, x, y)
```

2011

3: LLVM IR for device code (1/2)

```
; ModuleID = 'sincos.sincos_loop_1_kernelgen.cuda.device.F90.ir'
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 20110810 (prerelease) LLVM: 136347M\22"

define void @sincos_loop_1_kernelgen_(i32* nocapture %nz, i32* nocapture %ny, i32* nocapture %nx, [0
x float]* %xy, [0 x float]* %x, [0 x float]* %y) nounwind uwtable {
entry:
  %memtmp = alloca i32, align 4
  %memtmp3 = alloca i32, align 4
  %memtmp4 = alloca i32, align 4
  %0 = load i32* %nx, align 4
  %1 = sext i32 %0 to i64
  %2 = icmp slt i64 %1, 0
  %3 = select i1 %2, i64 0, i64 %1
  %4 = load i32* %ny, align 4
  %5 = sext i32 %4 to i64
  %6 = mul nsw i64 %3, %5
  %7 = icmp slt i64 %6, 0
  %8 = select i1 %7, i64 0, i64 %6
  %not =
  %9 = su
  %10 =
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_z(i32* noalias %memtmp, i32 1,
i32 %10) nounwind
  %11 = load i32* %ny, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_y(i32* noalias %memtmp3, i32 1,
i32 %11) nounwind
```

```
call sincos_loop_1_kernelgen_blockidx_z(k, 1, nz)
```

2011

3: LLVM IR for device code (2/2)

```
  %12 = load i32* %nx, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_x(i32* noalias %memtmp4, i32
1, i32 %12) nounwind
  %13 = load i32* %memtmp4, align 4
  %14 = sext i32 %13 to i64
  %15 = load i32* %memtmp, align 4
  %16 = sext i32 %15 to i64
  %17 = mul nsw i64 %16, %8
  %18 = load i32* %memtmp3, align 4
  %19 = sext i32 %18 to i64
  %20 = mul nsw i64 %19, %3
  %21 = add i64 %14, %9
  %22 = add i64 %21, %17
  %23 = add i64 %22, %20
  %24 = getelementptr [0 x float]* %x, i64 0, i64 %23
  %25 = load float* %24, align 4
  %26 = call float @sinf(float %25) nounwind readnone
  %27 = getelementptr [0 x float]* %y, i64 0, i64 %23
  %28 = load float* %27, align 4
  %29 = call float @cosf(float %28) nounwind readnone
  %30 = fadd float %26, %29
  %31 = getelementptr [0 x float]* %xy, i64 0, i64 %23
  store float %30, float* %31, align 4
  ret void
}

declare void @sincos_loop_1_kernelgen_blockidx_z(i32* noalias, i32, i32, ...)
declare void @sincos_loop_1_kernelgen_blockidx_y(i32* noalias, i32, i32, ...)
declare void @sincos_loop_1_kernelgen_blockidx_x(i32* noalias, i32, i32, ...)
declare float @sinf(float) nounwind readnone
declare float @cosf(float) nounwind readnone
```

2011

3: LLVM IR for device code (2/2)

```
  %12 = load i32* %nx, align 4
  call void (i32*, i32, i32, ...)* @sincos_loop_1_kernelgen_blockidx_x(i32* noalias %memtmp4, i32
1, i32 %12) nounwind
  %13 = load i32* %memtmp4, align 4
  %14 = sext i32 %13 to i64
  %15 = load i32* %memtmp, align 4
  %16 = sext i32 %15 to i64
  %17 = mul nsw i64 %16, %8
  %18 = load i32* %memtmp3, align 4
  %19 = sext i32 %18 to i64
  %20 = mul nsw i64 %19, %3
  %21 = a
  %22 = a
  %23 = a
  %24 = getelementptr [0 x float]* %x, i64 0, i64 %23
  %25 = load float* %24, align 4
  %26 = call float @sinf(float %25) nounwind readnone
  %27 = getelementptr [0 x float]* %y, i64 0, i64 %23
  %28 = load float* %27, align 4
  %29 = call float @cosf(float %28) nounwind readnone
  %30 = fadd float %26, %29
  %31 = getelementptr [0 x float]* %xy, i64 0, i64 %23
  store float %30, float* %31, align 4
  ret void
}

declare void @sincos_loop_1_kernelgen_blockidx_z(i32* noalias, i32, i32, ...)
declare void @sincos_loop_1_kernelgen_blockidx_y(i32* noalias, i32, i32, ...)
declare void @sincos_loop_1_kernelgen_blockidx_x(i32* noalias, i32, i32, ...)
declare float @sinf(float) nounwind readnone
declare float @cosf(float) nounwind readnone
```

```
xy(i, j, k) = sin(x(i, j, k)) + cos(y(i, j, k))
```

2011

4: C code for LLVM IR (1/3)

```c
void sincos_loop_1_kernelgen_(
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
unsigned int *llvm_cbe_nz,
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
unsigned int *llvm_cbe_ny,
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
unsigned int *llvm_cbe_nx,
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
l_unnamed_0 (*llvm_cbe_xy),
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
l_unnamed_0 (*llvm_cbe_x),
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
l_unnamed_0 (*llvm_cbe_y)) {
  unsigned int llvm_cbe_memtmp;    /* Address-exposed local */
  unsigned int llvm_cbe_memtmp3;    /* Address-exposed local */
  unsigned int llvm_cbe_memtmp4;    /* Address-exposed local */
  unsigned int llvm_cbe_tmp__1;
  unsigned long long llvm_cbe_tmp__2;
  unsigned long long llvm_cbe_tmp__3;
```

2011

4: C code for LLVM IR (1/3)

```
void sincos_loop_1_kernelgen_(
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
unsigned int *llvm_cbe_nz,
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
unsigned int *llvm_cbe_ny,
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
unsigned int *llvm_cbe_nx,
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
l_unnamed_0 (*llvm_cbe_xy),
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
l_unnamed_0 (*llvm_cbe_x),
#ifdef __OPENCL_DEVICE_FUNC__
__global
#endif // __OPENCL_DEVICE_FUNC__
l_unnamed_0 (*llvm_cbe_y)) {
  unsigned int llvm_cbe_memtmp;    /* Address-exposed local */
  unsigned int llvm_cbe_memtmp3;    /* Address-exposed local */
  unsigned int llvm_cbe_memtmp4;    /* Address-exposed local */
  unsigned int llvm_cbe_tmp__1;
  unsigned long long llvm_cbe_tmp__2;
  unsigned long long llvm_cbe_tmp__3;
```

In case of OpenCL, add __global attribute to subroutine arguments

4: C code for LLVM IR (2/3)

```
  unsigned int llvm_cbe_tmp__4;
  unsigned long long llvm_cbe_tmp__5;
  unsigned long long llvm_cbe_tmp__6;
  unsigned int llvm_cbe_tmp__7;
  unsigned int llvm_cbe_tmp__8;
  unsigned int llvm_cbe_tmp__9;
  unsigned int llvm_cbe_tmp__10;
  unsigned int llvm_cbe_tmp__11;
  unsigned int llvm_cbe_tmp__12;
  unsigned long long llvm_cbe_tmp__13;
  float llvm_cbe_tmp__14;
  float llvm_cbe_tmp__15;
  float llvm_cbe_tmp__16;
  float llvm_cbe_tmp__17;

  llvm_cbe_tmp__1 = *llvm_cbe_nx;
  llvm_cbe_tmp__2 = ((signed long long )(signed int )llvm_cbe_tmp__1);
  llvm_cbe_tmp__3 = (((((signed long long )llvm_cbe_tmp__2) < ((signed long long )0ull))) ?
(0ull) : (llvm_cbe_tmp__2));
  llvm_cbe_tmp__4 = *llvm_cbe_ny;
  llvm_cbe_tmp__5 = ((unsigned long long )(((unsigned long long )llvm_cbe_tmp__3) * ((unsigned long
long )(((signed long long )(signed int )llvm_cbe_tmp__4)))));
  llvm_cbe_tmp__6 = (((((signed long long )llvm_cbe_tmp__5) < ((signed long long )0ull))) ?
(0ull) : (llvm_cbe_tmp__5));
  llvm_cbe_tmp__7 = *llvm_cbe_nz;
  sincos_loop_1_kernelgen_blockidx_z((&llvm_cbe_memtmp), 1u, llvm_cbe_tmp__7);
  llvm_cbe_tmp__8 = *llvm_cbe_ny;
  sincos_loop_1_kernelgen_blockidx_y((&llvm_cbe_memtmp3), 1u, llvm_cbe_tmp__8);
  llvm_cbe_tmp__9 = *llvm_cbe_nx;
  sincos_loop_1_kernelgen_blockidx_x((&llvm_cbe_memtmp4), 1u, llvm_cbe_tmp__9);
```

2011

4: C code for LLVM IR (2/3)

```
  unsigned int llvm_cbe_tmp__4;
  unsigned long long llvm_cbe_tmp__5;
  unsigned long long llvm_cbe_tmp__6;
  unsigned int llvm_cbe_tmp__7;
  unsigned int llvm_cbe_tmp__8;
  unsigned int llvm_cbe_tmp__9;
  unsigned int llvm_cbe_tmp__10;
  unsigned int llvm_cbe_tmp__11;
  unsigned int llvm_cbe_tmp__12;
  unsigned long long llvm_cbe_tmp__13;
  float llvm_cbe_tmp__14;
  float llvm_cbe_tmp__15;
  float llvm_cbe_tmp__16;
  float llvm_cbe_tmp__17;

  llvm_cbe_tmp__1 = *llvm_cbe_nx;
  llvm_cbe_tmp__2 = ((signed long long )(signed int )llvm_cbe_tmp__1);
  llvm_cbe_tmp__3 = (((((signed long long )llvm_cbe_tmp__2) < ((signed long long )0ull))) ?
(0ull) : (llvm_cbe_tmp__2));
  llvm_cbe_tmp__4 = *llvm_cbe_ny;
  llvm_cbe_tmp__5 = ((unsigned long long )(((unsigned long long )llvm_cbe_tmp__3) * ((unsigned long
long )(((signed long long )(signed int )llvm_cbe_tmp__4)))));
  llvm_cb
(0ull) :
  llvm_cb
  sincos_loop_1_kernelgen_blockidx_z((&llvm_cbe_memtmp), 1u, llvm_cbe_tmp__7);
  llvm_cbe_tmp__8 = *llvm_cbe_ny;
  sincos_loop_1_kernelgen_blockidx_y((&llvm_cbe_memtmp3), 1u, llvm_cbe_tmp__8);
  llvm_cbe_tmp__9 = *llvm_cbe_nx;
  sincos_loop_1_kernelgen_blockidx_x((&llvm_cbe_memtmp4), 1u, llvm_cbe_tmp__9);
```

```
call sincos_loop_1_kernelgen_blockidx_z(k, 1, nz)
```

2011

4: C code for LLVM IR (3/3)

```
  llvm_cbe_tmp__10 = *(&llvm_cbe_memtmp4);
  llvm_cbe_tmp__11 = *(&llvm_cbe_memtmp);
  llvm_cbe_tmp__12 = *(&llvm_cbe_memtmp3);
  llvm_cbe_tmp__13 = ((unsigned long long )(((unsigned long long )(((unsigned long long )
(((unsigned long long )(((unsigned long long )(((unsigned long long )(((signed long long )(signed
int )llvm_cbe_tmp__10))) + ((unsigned long long )(((unsigned long long )(((unsigned long long )
(llvm_cbe_tmp__3 ^ 18446744073709551615ull)) - ((unsigned long long )llvm_cbe_tmp__6)))))))))) +
((unsigned long long )(((unsigned long long )(((unsigned long long )(((signed long long )(signed
int )llvm_cbe_tmp__11))) * ((unsigned long long )llvm_cbe_tmp__6)))))))))) + ((unsigned long long )
(((unsign                                                                    p__12)))
* ((unsig
  llvm_cb
  llvm_cbe_tmp__15 = sinf(llvm_cbe_tmp__14);
  llvm_cbe_tmp__16 = *((&(*llvm_cbe_y).array[((signed long long )llvm_cbe_tmp__13)]));
  llvm_cbe_tmp__17 = cosf(llvm_cbe_tmp__16);
  *((&(*llvm_cbe_xy).array[((signed long long )llvm_cbe_tmp__13)])) = (((float )(llvm_cbe_tmp__15 +
llvm_cbe_tmp__17)));
  return;
}
```

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

2011

Compiling sincos

```
[marcusmae@noisy ~]$ cd Programming/kernelgen/trunk/tests/performance/sincos/
[marcusmae@noisy sincos]$ make 32/sincos
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl,  -O3 -g -c sincos.f90 -o 32/sincos.o
kernelgen   >> sincos.f90:42: portable 3-dimensional loop
kernelgen   >> sincos.f90:42: selecting this loop
c   >> ptxas info    : Compiling entry function 'sincos_loop_1_kernelgen_cuda'
for 'sm_20'
c   >> ptxas info    : Function properties for sincos_loop_1_kernelgen_cuda
c   >>     56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
c   >> ptxas info    : Used 12 registers, 4+0 bytes lmem, 56 bytes cmem[0], 24
bytes cmem[2], 44 bytes cmem[16]
/usr/bin/gcc -I/home/marcusmae/Programming/kernelgen/trunk/include
-I/home/marcusmae/opt/kgen/include -m32 -O3 -g -std=c99 -I/opt/kgen/include -c
main.c -o 32/main.o
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl, 32/main.o 32/sincos.o -o 32/sincos
```

2011

Compiling sincos

```
[marcusmae@noisy ~]$ cd Programming/kernelgen/trunk/tests/performance/sincos/
[marcusmae@noisy sincos]$ make 32/sincos
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl,  -O3 -g -c sincos.f90 -o 32/sincos.o
kernelge
kernelge
c   >> p                                                        gen_cuda'
for 'sm_20'
c   >> ptxas info    : Function properties for sincos_loop_1_kernelgen_cuda
c   >>     56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
c   >> ptxas info    : Used 12 registers, 4+0 bytes lmem, 56 bytes cmem[0], 24
bytes cmem[2], 44 bytes cmem[16]
/usr/bin/gcc -I/home/marcusmae/Programming/kernelgen/trunk/include
-I/home/marcusmae/opt/kgen/include -m32 -O3 -g -std=c99 -I/opt/kgen/include -c
main.c -o 32/main.o
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl, 32/main.o 32/sincos.o -o 32/sincos
```

KernelGen compilation command, specifying target devices and compilers to use

Compiling sincos

```
[marcusmae@noisy ~]$ cd Programming/kernelgen/trunk/tests/performance/sincos/
[marcusmae@noisy sincos]$ make 32/sincos
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl,  -O3 -g -c sincos.f90 -o 32/sincos.o
kernelgen   >> sincos.f90:42: portable 3-dimensional loop
kernelgen   >> sincos.f90:42: selecting this loop
c   >> p                                                            gen_cuda'
for 'sm_
c   >> p                                                            n_cuda
c   >>     56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
c   >> ptxas info    : Used 12 registers, 4+0 bytes lmem, 56 bytes cmem[0], 24
bytes cmem[2], 44 bytes cmem[16]
/usr/bin/gcc -I/home/marcusmae/Programming/kernelgen/trunk/include
-I/home/marcusmae/opt/kgen/include -m32 -O3 -g -std=c99 -I/opt/kgen/include -c
main.c -o 32/main.o
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl, 32/main.o 32/sincos.o -o 32/sincos
```

KernelGen reports indentified portable loops and those of them selected to have device version

2011

Compiling sincos

```
[marcusmae@noisy ~]$ cd Programming/kernelgen/trunk/tests/performance/sincos/
[marcusmae@noisy sincos]$ make 32/sincos
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl,  -O3 -g -c sincos.f90 -o 32/sincos.o
kernelgen   >> sincos.f90:42: portable 3-dimensional loop
kernelgen   >> sincos.f90:42: selecting this loop
c   >> ptxas info    : Compiling entry function 'sincos_loop_1_kernelgen_cuda'
for 'sm_20'
c   >> ptxas info    : Function properties for sincos_loop_1_kernelgen_cuda
c   >>      56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
c   >> ptxas info    : Used 12 registers, 4+0 bytes lmem, 56 bytes cmem[0], 24
bytes cmem[2], 44 bytes cmem[16]
/usr/bin/gcc -I/home/marcus                          trunk/include
-I/home/marcusmae/opt/kgen/                          99 -I/opt/kgen/include -c
main.c -o 32/main.o
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl, 32/main.o 32/sincos.o -o 32/sincos
```

Output from ptx-as

Compiling sincos

```
[marcusmae@noisy ~]$ cd Programming/kernelgen/trunk/tests/performance/sincos/
[marcusmae@noisy sincos]$ make 32/sincos
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl,  -O3 -g -c sincos.f90 -o 32/sincos.o
kernelgen   >> sincos.f90:42: portable 3-dimensional loop
kernelgen   >> sincos.f90:42: selecting this loop
c   >> ptxas info    : Compiling entry function 'sincos_loop_1_kernelgen_cuda'
for 'sm_20'
c   >> ptxas info    : Function properties for sincos_loop_1_kernelgen_cuda
c   >>     56 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
c   >> ptxas info    : Used 12 registers, 4+0 bytes lmem, 56 bytes cmem[0], 24
bytes cmem[2], 44 bytes cmem[16]
/usr/bin/gcc -I/home/marcusmae/Programming/kernelgen/trunk/include
-I/home/marcusmae/opt/kgen/                    -std=c99 -I/opt/kgen/include -c
main.c -o 32/main.o
```

Linker command

```
kgen-gfortran -Wk,--host-compiler=/usr/bin/gfortran
-I/home/marcusmae/Programming/kernelgen/trunk/include -Wk,--cpu-compiler=gcc
-Wk,--cuda-compiler=nvcc -m32 -Wk,--opencl-compiler=kgen-opencl-embed -Wk,--
kernel-target=cpu,cuda,opencl, 32/main.o 32/sincos.o -o 32/sincos
```

Testing sincos

```
# By default – execute on CPU
[marcusmae@noisy sincos]$ 64/sincos 128 128 4 4 4 16
gpu time = 0.663560 sec
cpu time = 0.642710 sec
max diff = 0.000000e+00 @ 0

# Set default runmode to 2 to execute CUDA versions of all kernels
[marcusmae@noisy sincos]$ kernelgen_runmode=2 sincos_loop_1_kernelgen=2
64/sincos 128 128 4 4 4 16
gpu time = 0.369378 sec
cpu time = 0.643688 sec
max diff = 2.384186e-07 @ 310

# Set default runmode to 4 to execute OpenCL versions of all
kernels
[marcusmae@noisy sincos]$ kernelgen_runmode=4 sincos_loop_1_kernelgen=4
32/sincos 128 128 4 4 4 16
gpu time = 0.443252 sec
cpu time = 1.134842 sec
max diff = 2.384186e-07 @ 4349389
```

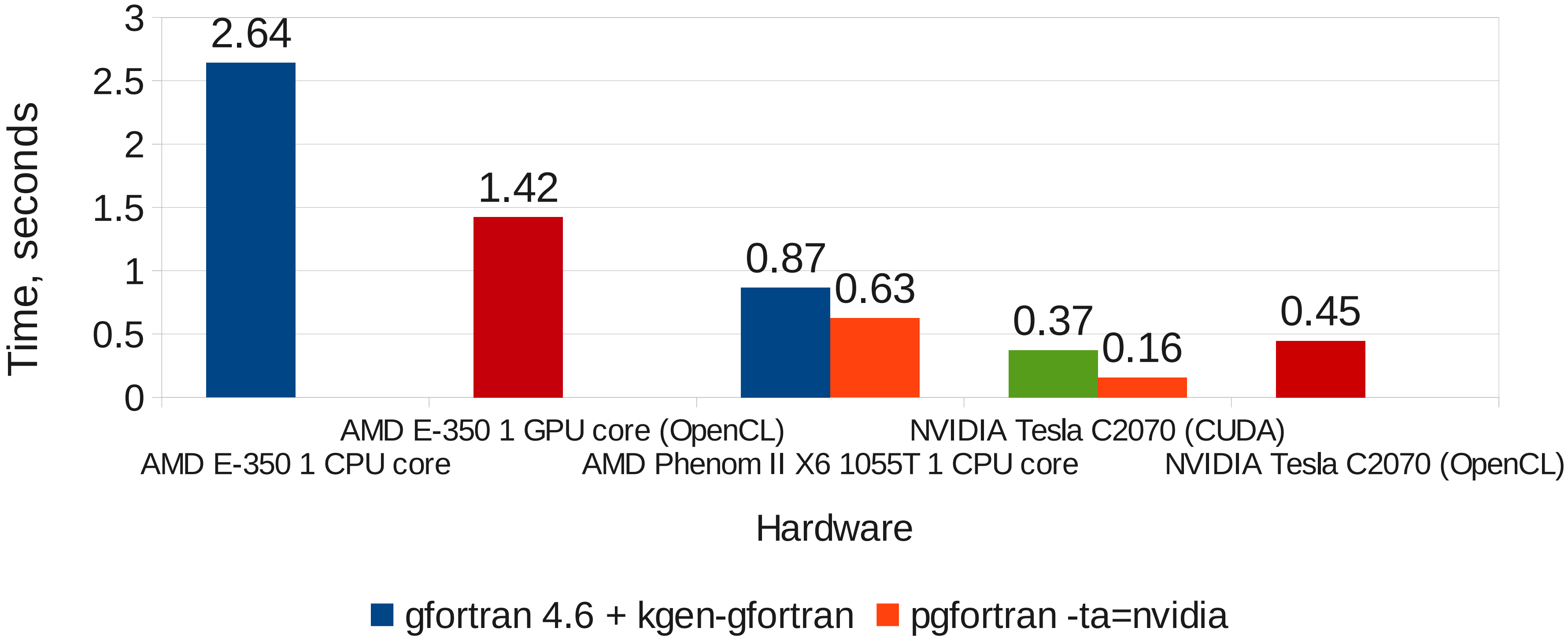
Testing sincos

```
# Add debug output filter bits to show more info
[marcusmae@noisy sincos]$ kernelgen_debug_output=11 kernelgen_runmode=2 kgen/32/sincos 512 512 64 1 1 1
launch.c:70 kernelgen message (debug) Launching sincos_loop_1_kernelgen_cuda for device NVIDIA Corporation:0
runmode "cuda"
parse_args.h:69 kernelgen message (debug) arg "unknown" ref = 0xfff9837c, size = 4, desc = 0xfff9837c
parse_args.h:69 kernelgen message (debug) arg "unknown" ref = 0xfff98378, size = 4, desc = 0xfff98378
parse_args.h:69 kernelgen message (debug) arg "unknown" ref = 0xfff98374, size = 4, desc = 0xfff98374
parse_args.h:69 kernelgen message (debug) arg "unknown" ref = 0xe3443008, size = 67108864, desc = 0xe3443008
parse_args.h:69 kernelgen message (debug) arg "unknown" ref = 0xf3447008, size = 67108864, desc = 0xf3447008
parse_args.h:69 kernelgen message (debug) arg "unknown" ref = 0xeb445008, size = 67108864, desc = 0xeb445008
map_cuda.h:107 kernelgen message (debug) symbol "unknown" maps memory segment [0xfff9837c .. 0xfff98380] to
[0x5400000 .. 0x5400004]
map_cuda.h:107 kernelgen message (debug) symbol "unknown" maps memory segment [0xfff98378 .. 0xfff9837c] to
[0x5400200 .. 0x5400204]
map_cuda.h:107 kernelgen message (debug) symbol "unknown" maps memory segment [0xfff98374 .. 0xfff98378] to
[0x5400400 .. 0x5400404]
map_cuda.h:107 kernelgen message (debug) symbol "unknown" maps memory segment [0xe3443008 .. 0xe7443008] to
[0x5500000 .. 0x9500000]
map_cuda.h:107 kernelgen message (debug) symbol "unknown" maps memory segment [0xf3447008 .. 0xf7447008] to
[0x9500000 .. 0xd500000]
map_cuda.h:107 kernelgen message (debug) symbol "unknown" maps memory segment [0xeb445008 .. 0xef445008] to
[0xd500000 .. 0x11500000]
gpu time = 0.373937 sec
cpu time = 1.136829 sec
max diff = 1.192093e-07 @ 17
```

4. Testing unoptimized generator

Example: sincos – performance

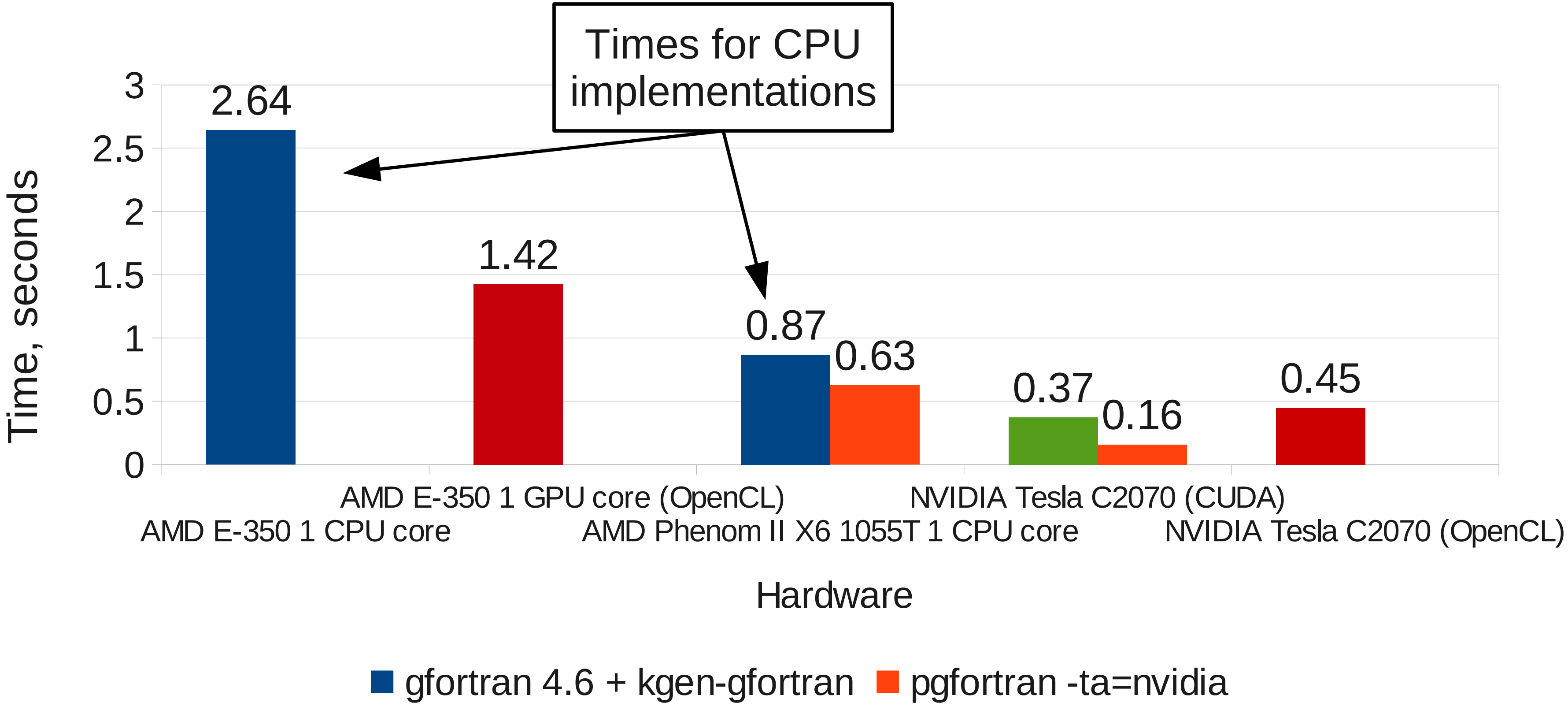
Performance of CPU binary generated by gfortran and CUDA kernel by KernelGen compared to host and device perfs, using PGI Accelerator 11.8 (orange)



2011

Example: sincos – performance

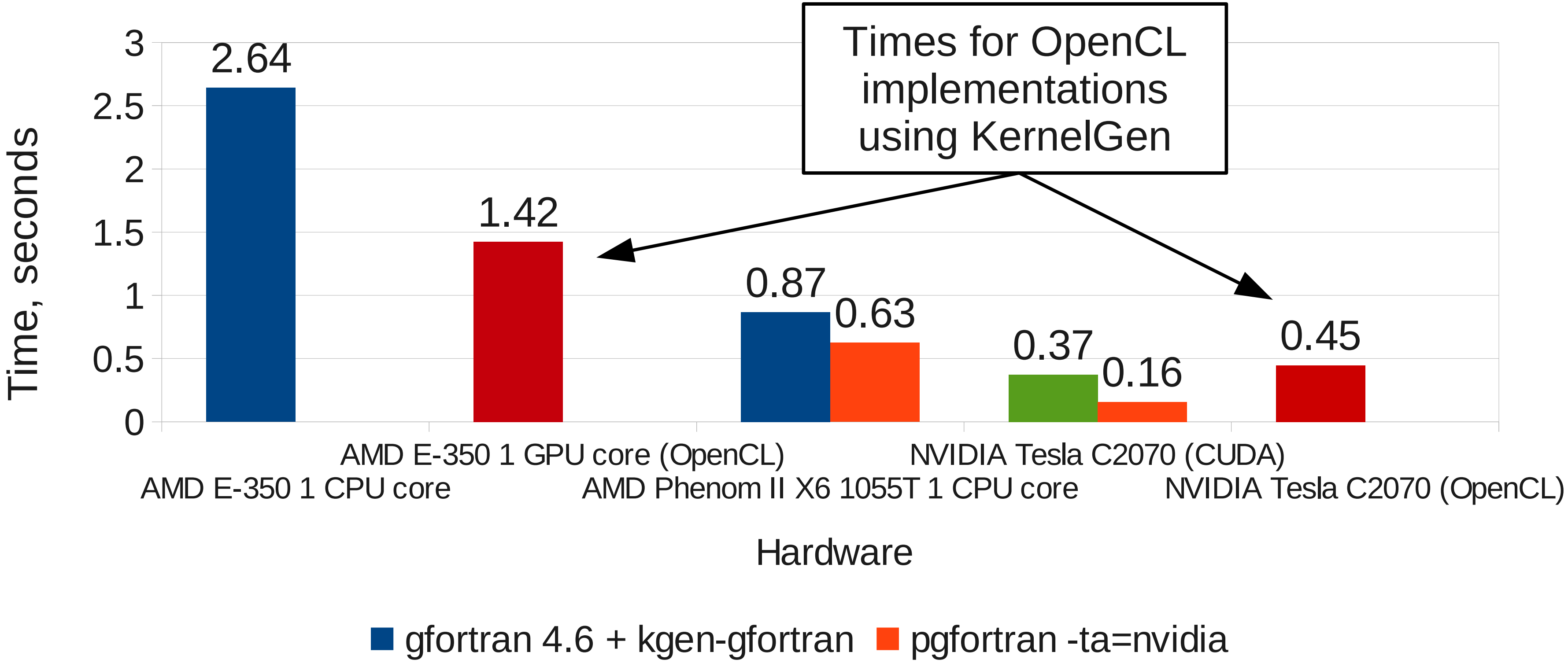
Performance of CPU binary generated by gfortran and CUDA kernel by KernelGen compared to host and device perfs, using PGI Accelerator 11.8 (orange)



2011

Example: sincos – performance

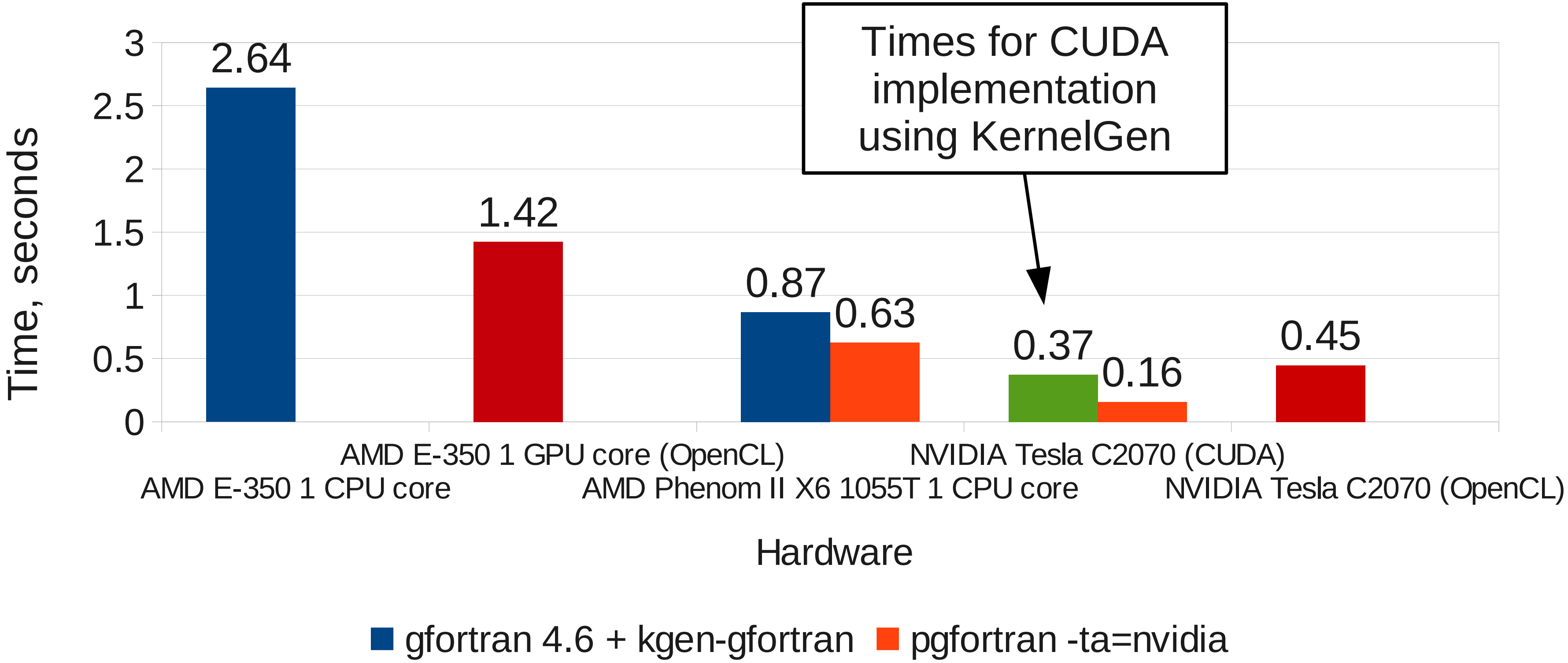
Performance of CPU binary generated by gfortran and CUDA kernel by KernelGen compared to host and device perfs, using PGI Accelerator 11.8 (orange)



2011

Example: sincos – performance

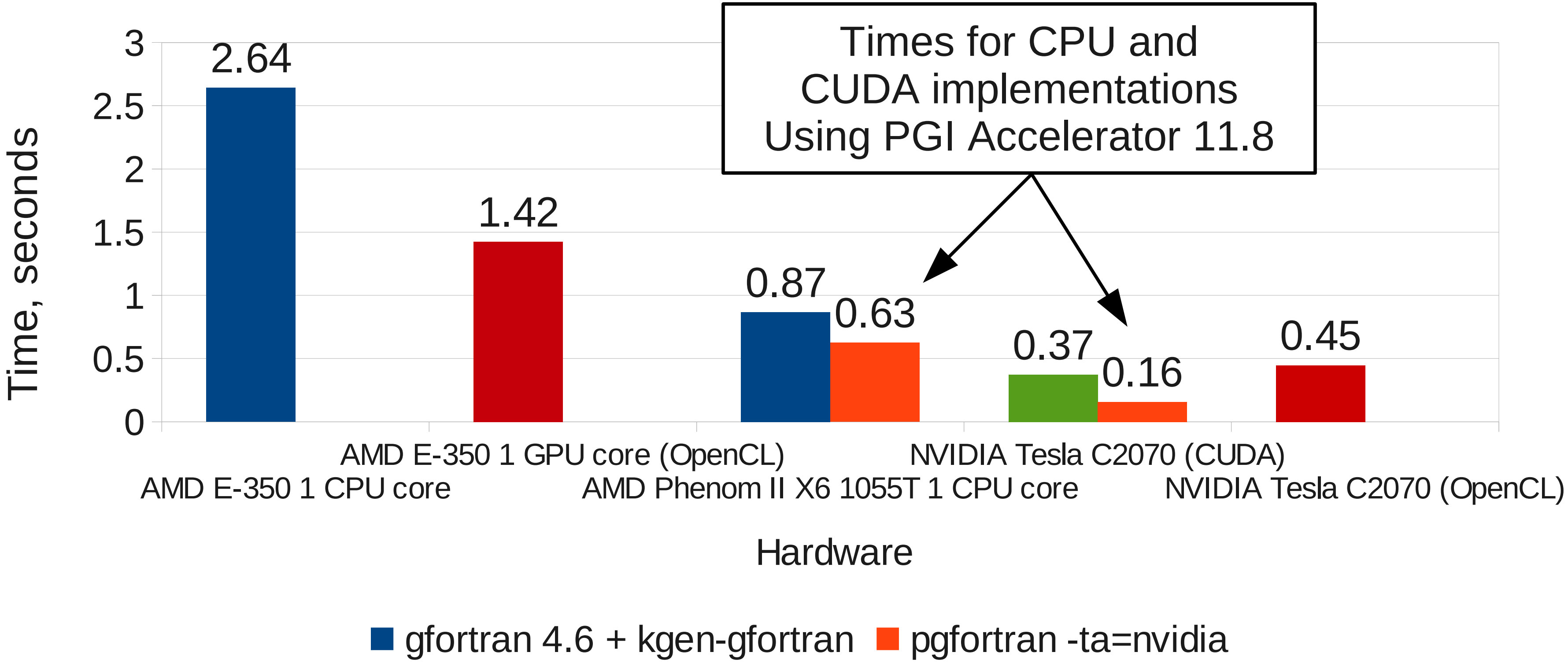
Performance of CPU binary generated by gfortran and CUDA kernel by KernelGen compared to host and device perfs, using PGI Accelerator 11.8 (orange)



2011

Example: sincos – performance

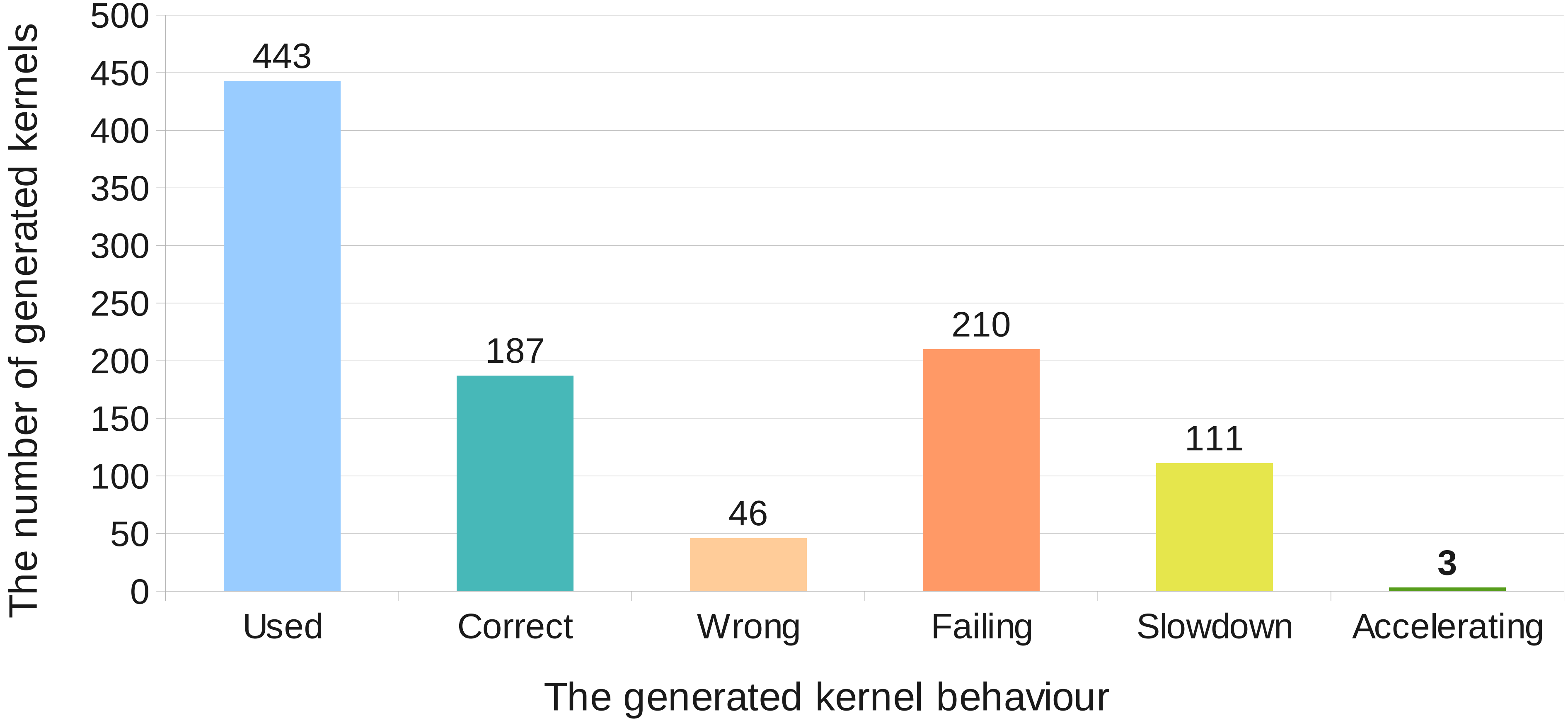
Performance of CPU binary generated by gfortran and OpenCL/CUDA kernels by KernelGen, compared to host and device perfs using PGI Accelerator 11.8 (orange)



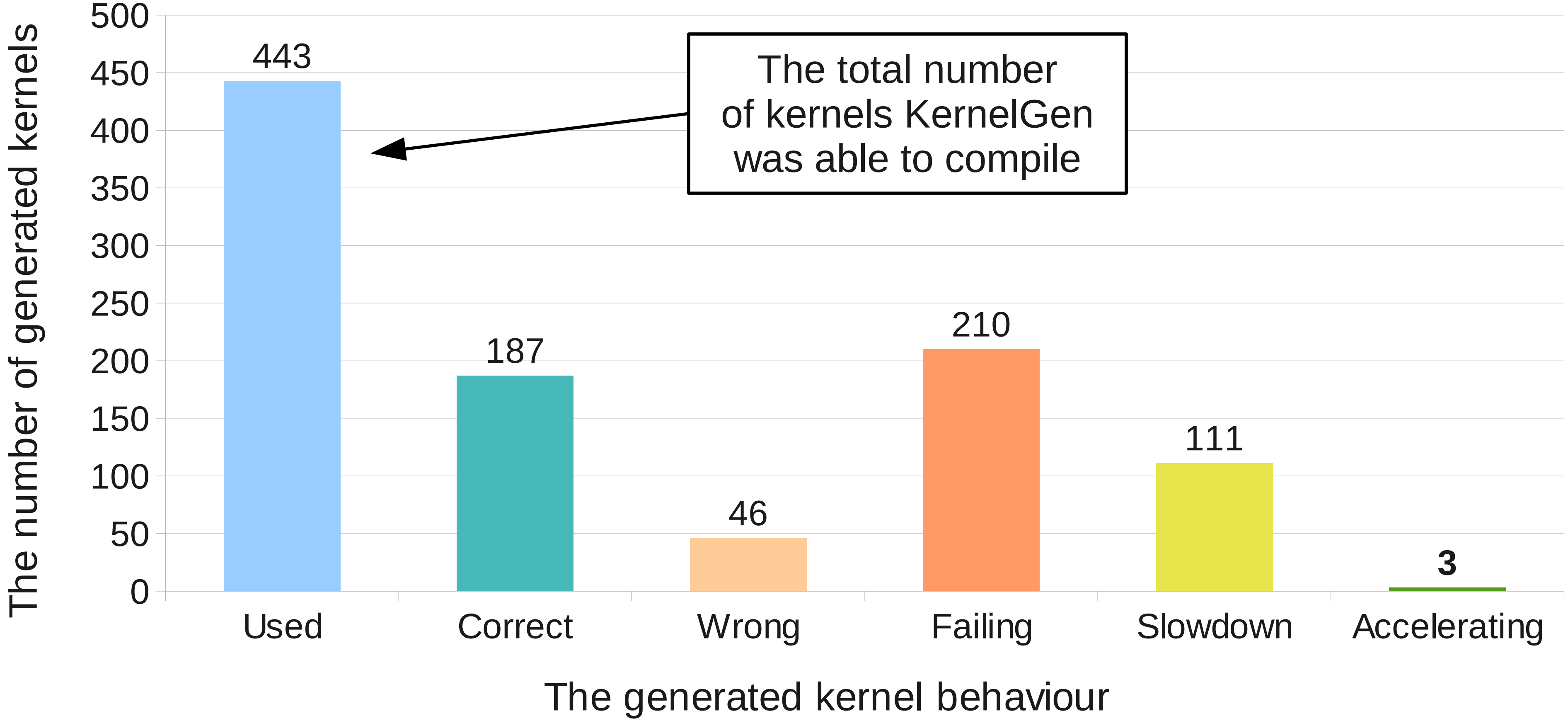
Times for CPU and CUDA implementations Using PGI Accelerator 11.8

2011

COSMO – coverage



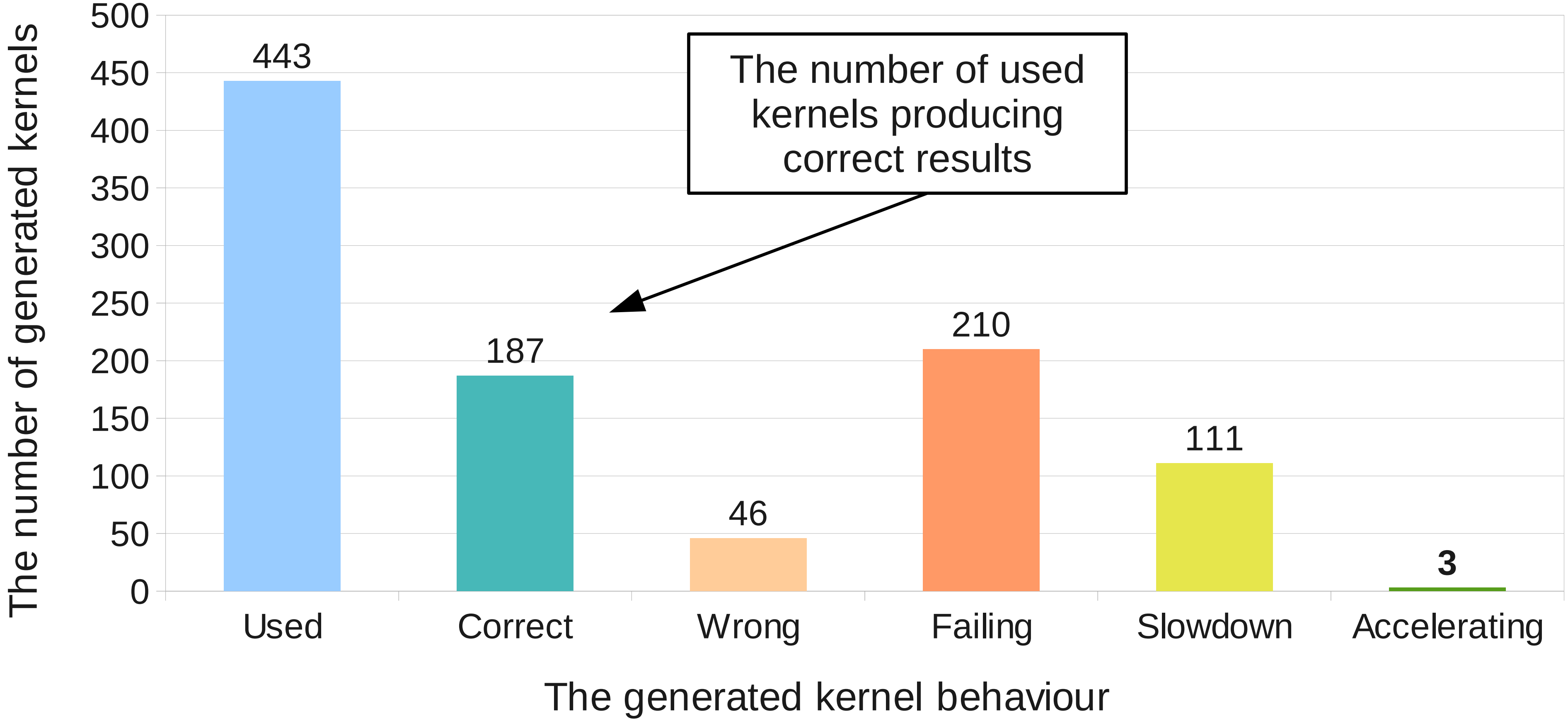
COSMO – coverage



The total number of kernels KernelGen was able to compile

2011

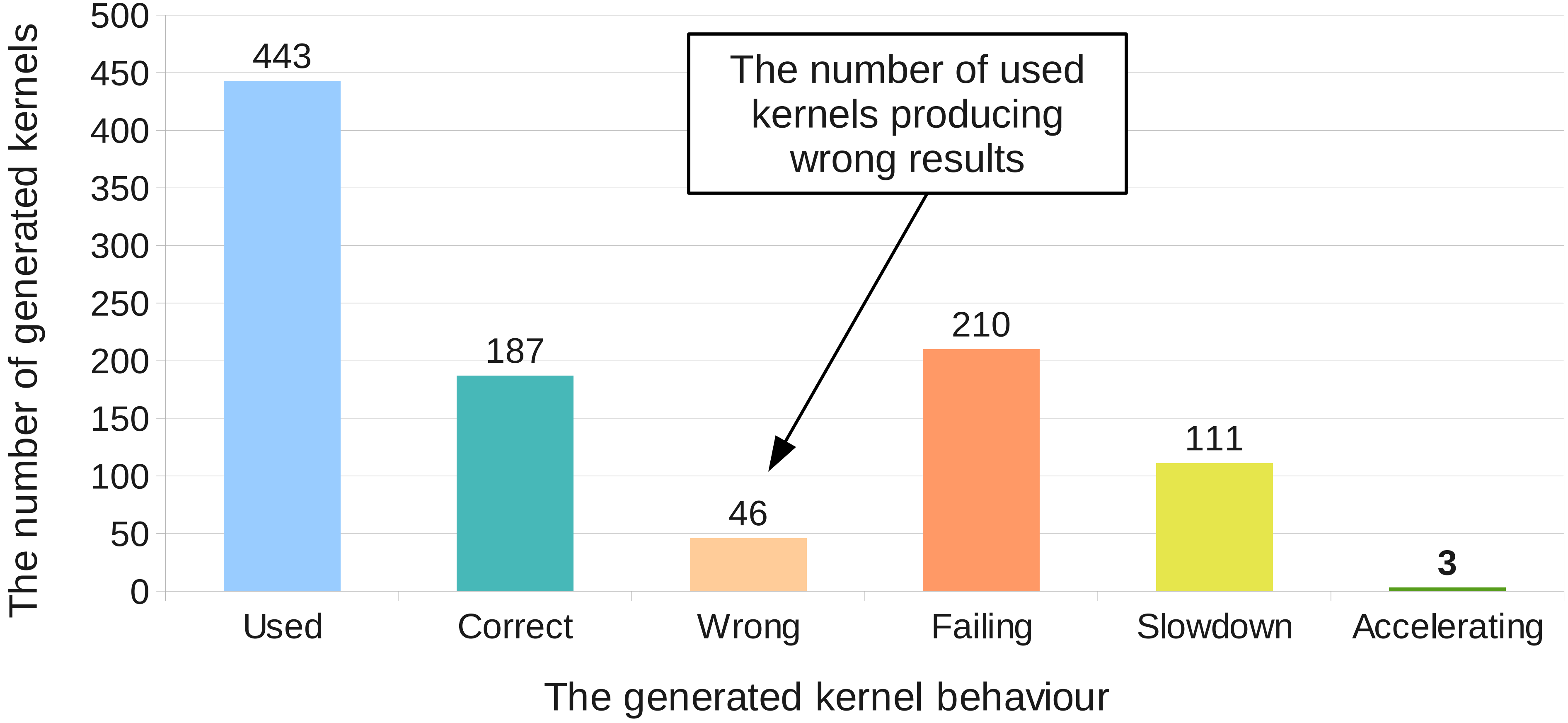
COSMO – coverage



The number of used kernels producing correct results

2011

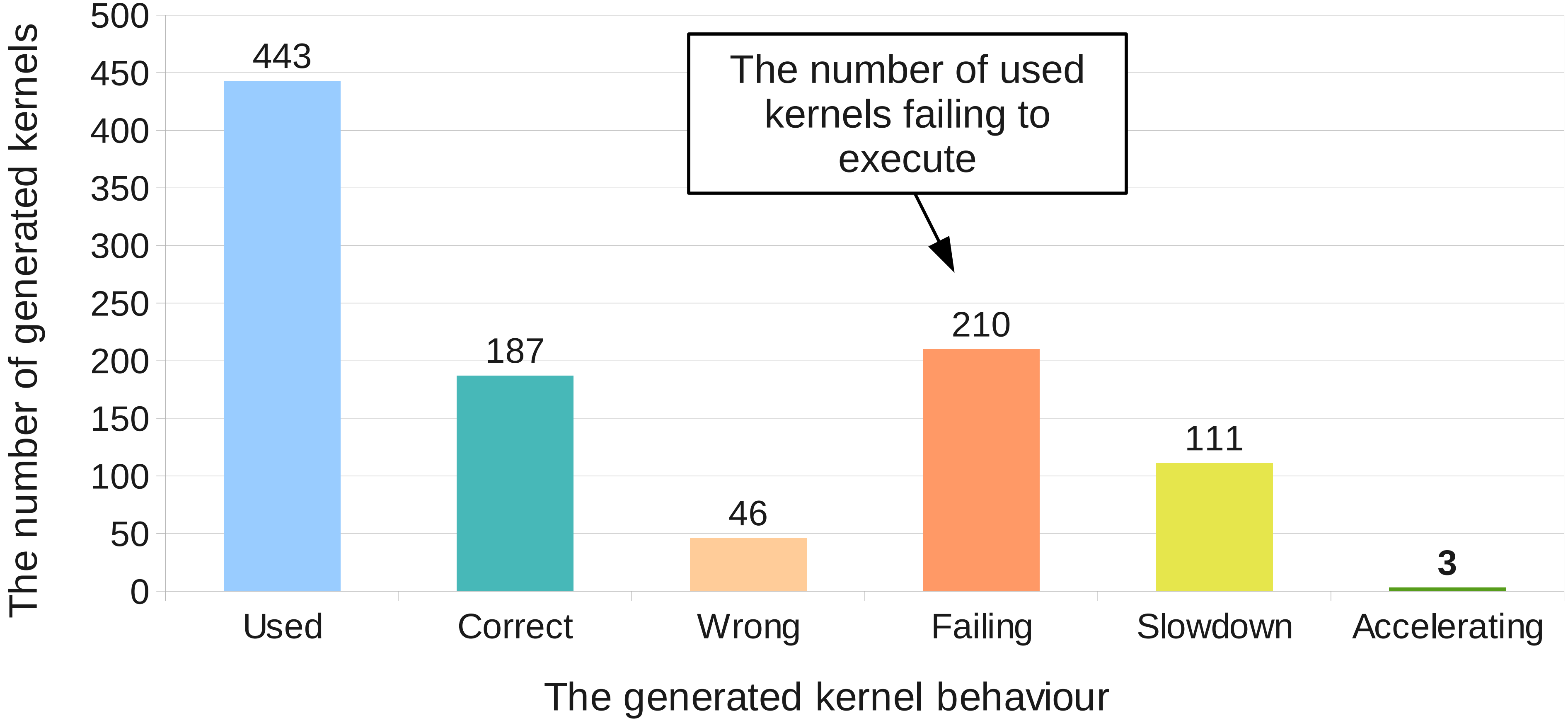
COSMO – coverage



The number of used kernels producing wrong results

2011

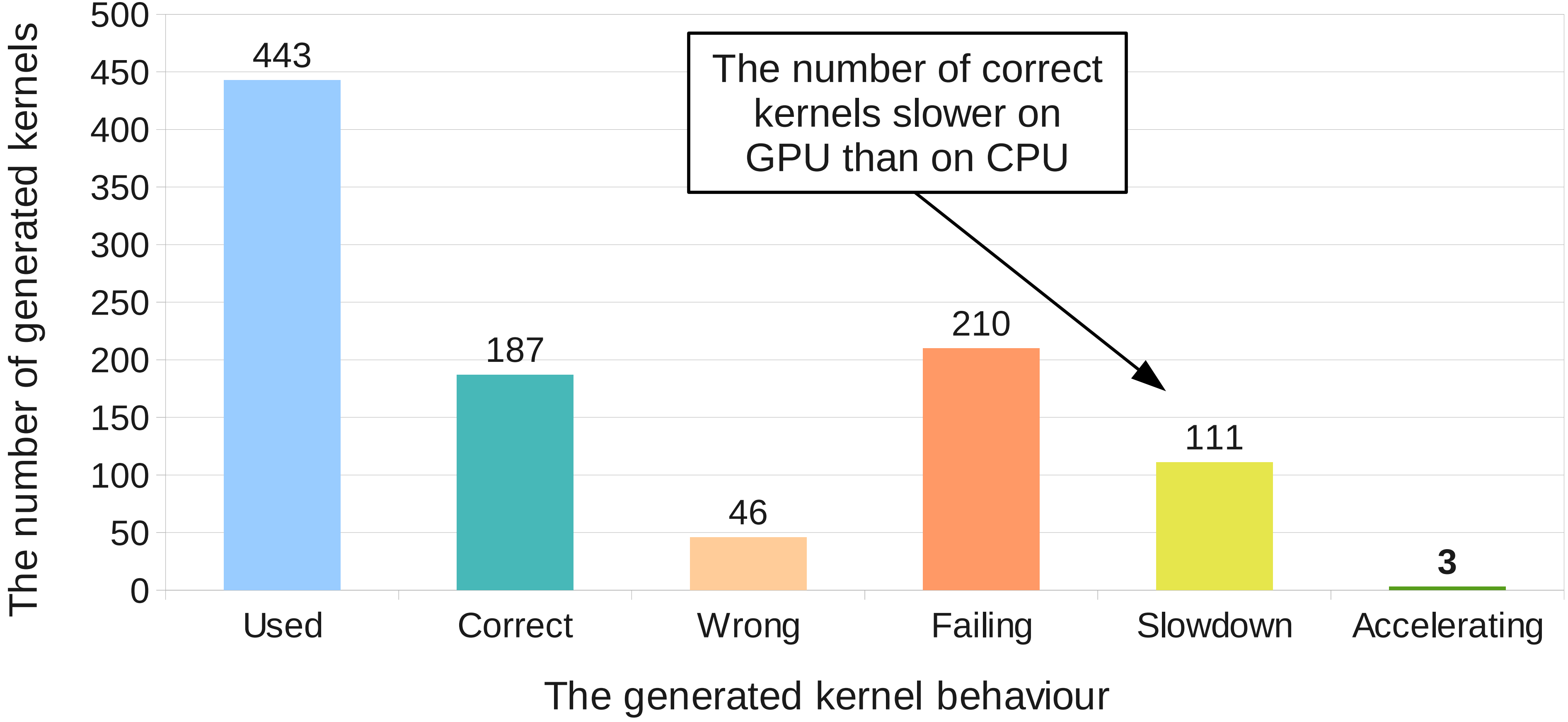
COSMO – coverage



The number of used kernels failing to execute

2011

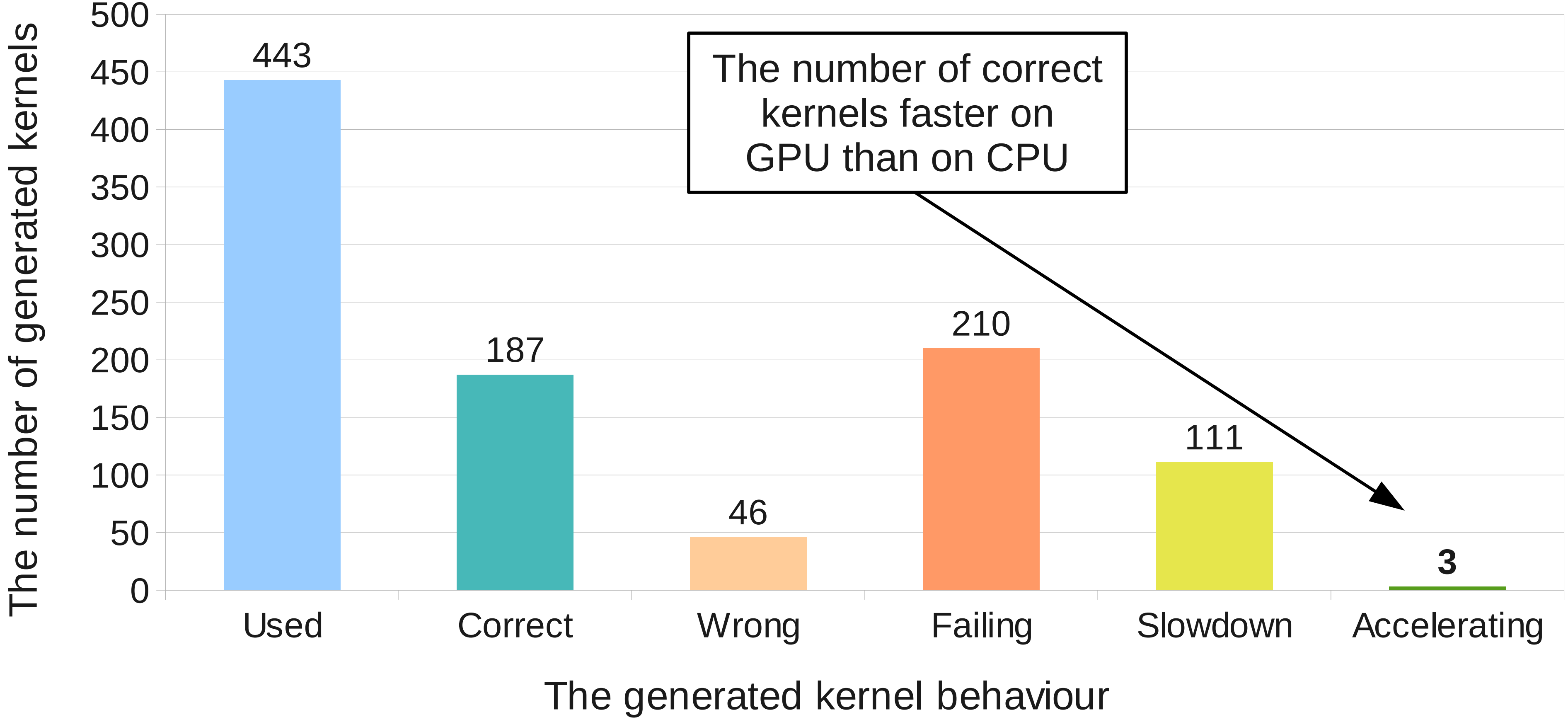
COSMO – coverage



The number of correct kernels slower on GPU than on CPU

2011

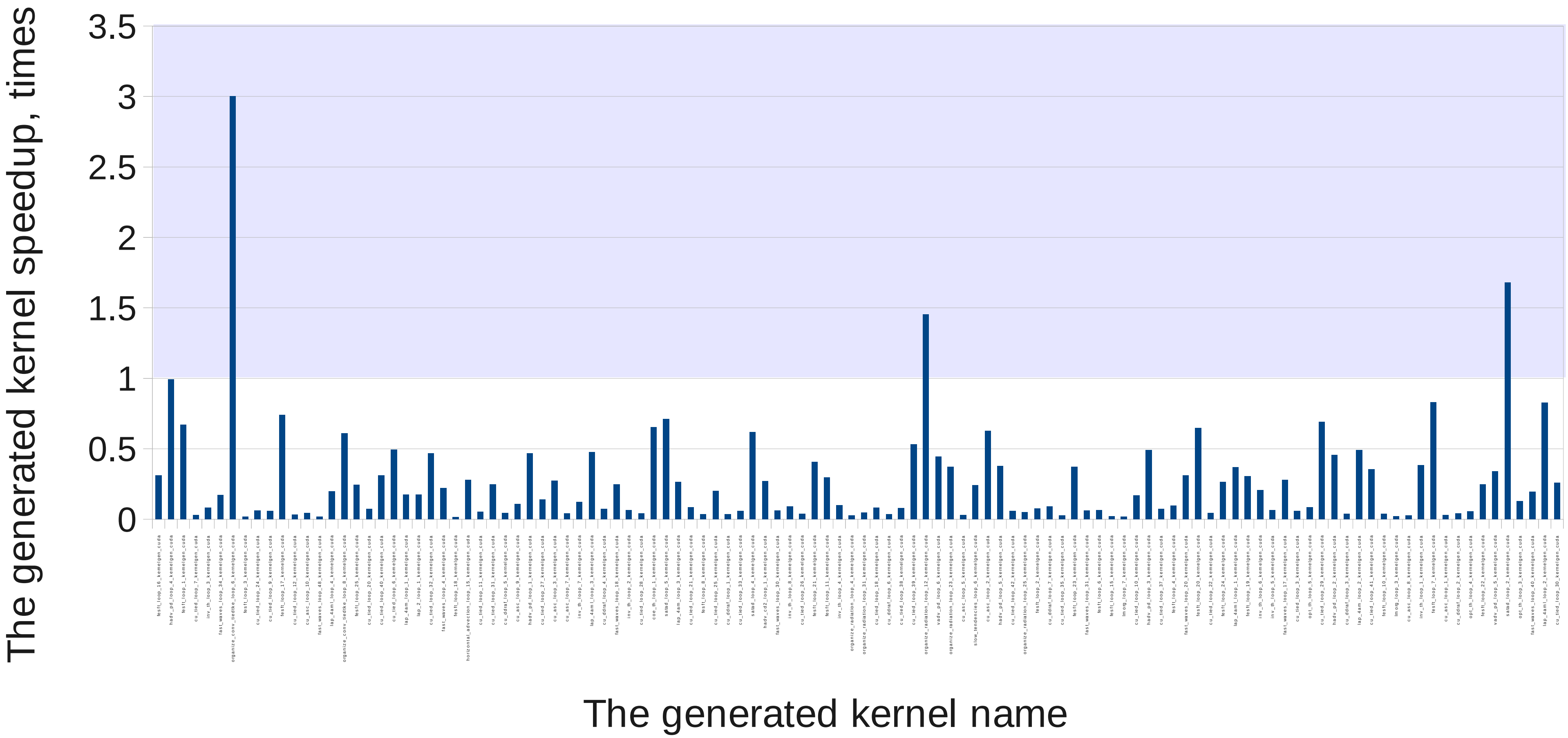
COSMO – coverage



The number of correct kernels faster on GPU than on CPU

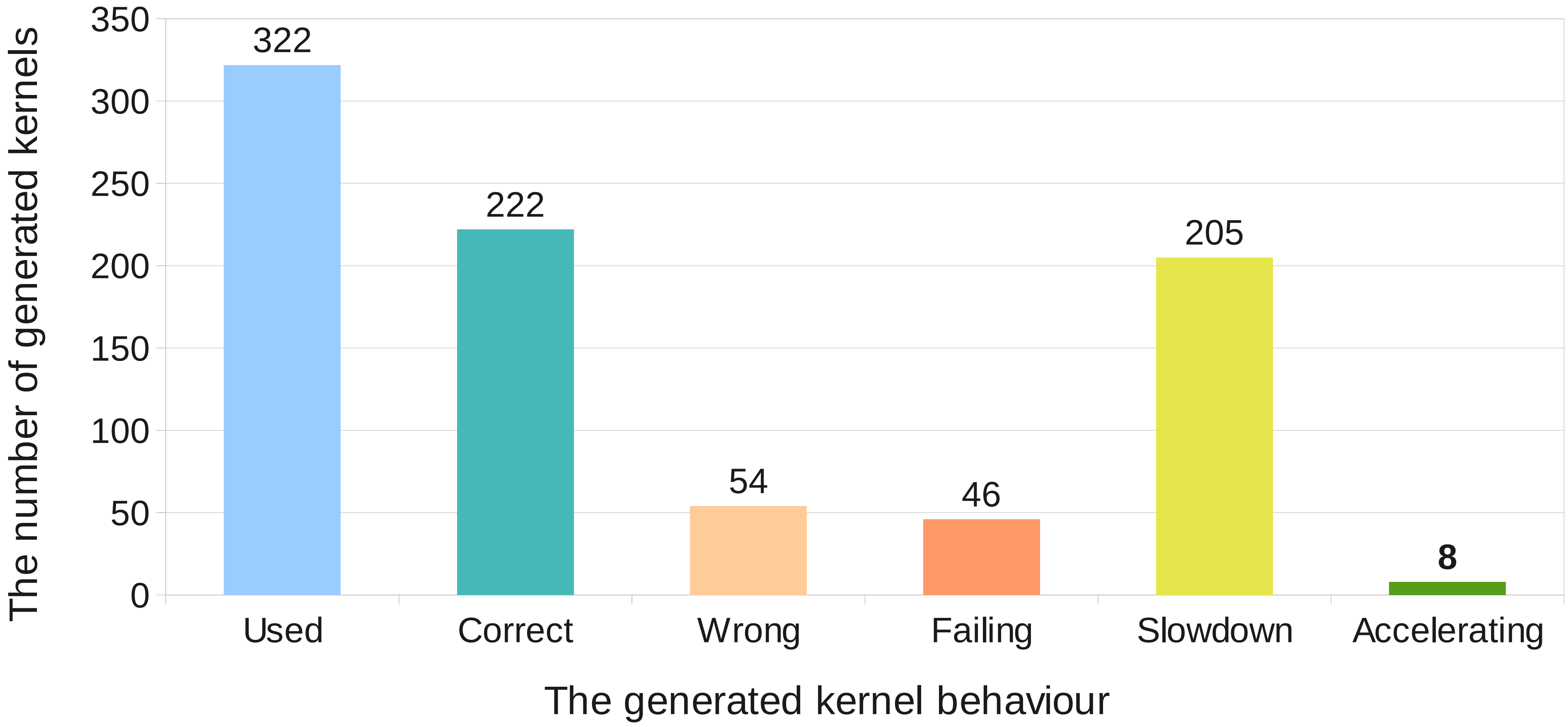
2011

COSMO – performance



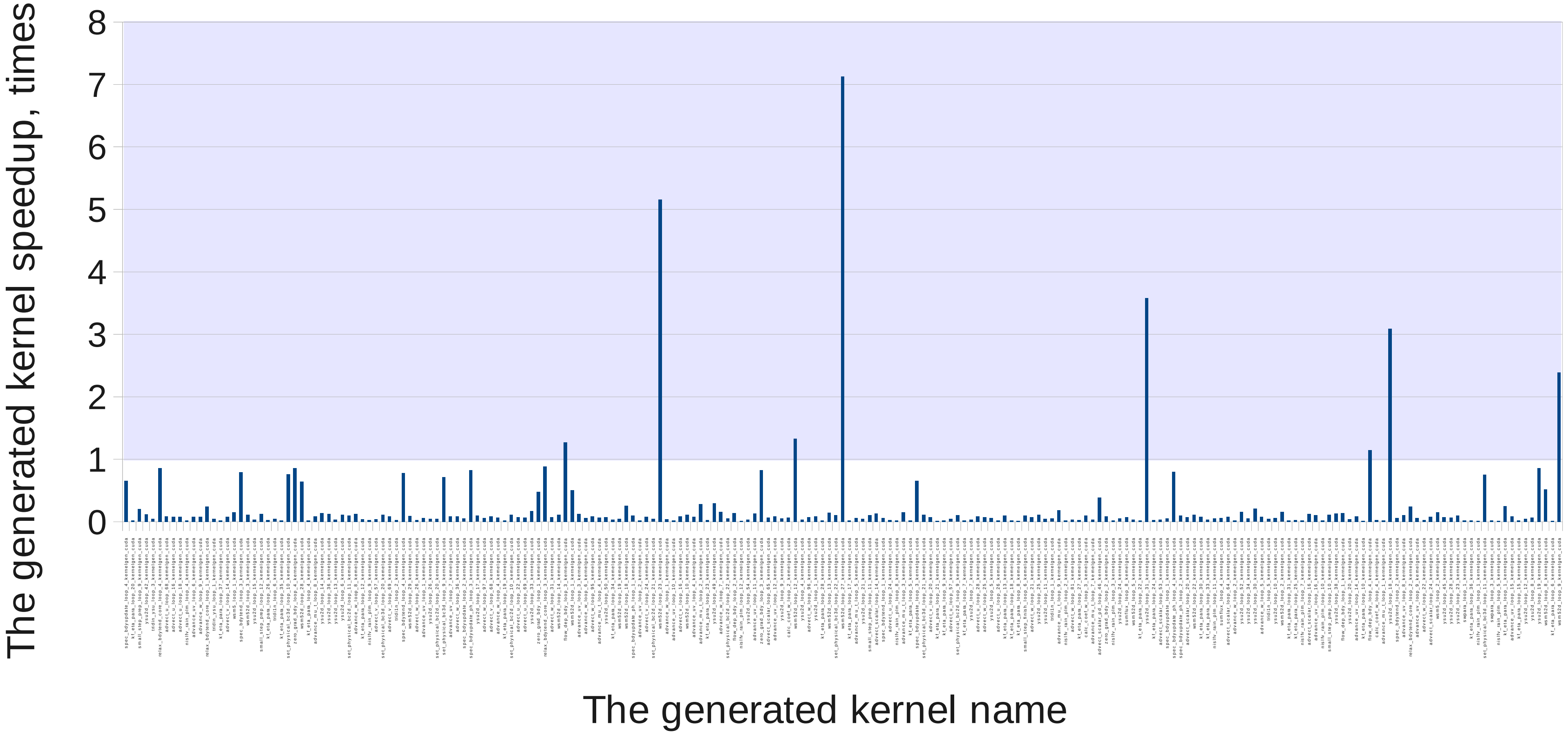
2011

WRF – coverage



2011

WRF – performance



2011

Why slowdown?

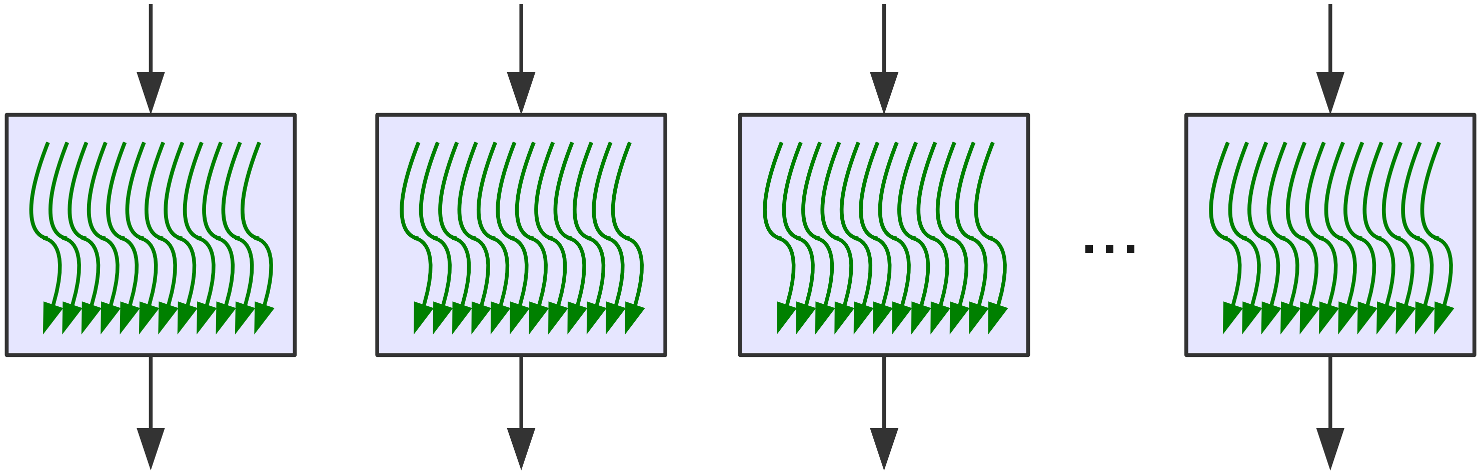
- KernelGen does not **yet** ultilize multiple threads inside thread blocks

```
# Performance of CUDA version with 4x4x16 threads grid and
copy-in/copy-out data masking is very similar to PGI's
[marcusmae@noisy sincos]$ 64/sincos 128 128 4 4 4 16 0
gpu time = 0.163355 sec
cpu time = 0.641843 sec
max diff = 2.384186e-07 @ 310

# Performance of CUDA version with 1x1x1 threads grid and
full data copying is very similar to KernelGen's
[marcusmae@noisy sincos]$ cuda/64/sincos 512 512 64 1 1 1 1
gpu time = 0.450221 sec
cpu time = 0.642180 sec
max diff = 2.384186e-07 @ 310
```

Why slowdown?

- KernelGen does not **yet** ultilize multiple threads inside thread blocks
- Threads in blocks would be possible with **tiling** optimization implemented (from LLVM/Polly)



2011

5. Development schedule

2011

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
- Complete existing host-device code split transform (previously started in 2009 for CellBE)
- Implement kernel invocation runtime
- Implement kernel self-checking runtime
- 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 threading and 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>

Stage 2 (July - October)

🟩 – done 🟪 – in progress now

- 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 threading and 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 carefully listen to feedback

2011

6. Team & resources

2011

Team

Artem Petrov	(testing, coordination)
Dr Yulia Martynova	(WRF testing)

Team

Artem Petrov	(testing, coordination)
Dr Yulia Martynova	(WRF testing)
Alexander Myltsev	(development, testing)
Dmitry Mikushin	(development, planning)

Team

Artem Petrov	(testing, coordination)
Dr Yulia Martynova	(WRF testing)

Alexander Myltsev	(development, testing)
Dmitry Mikushin	(development, planning)

Support from communities:

LLVM

Polly/LLVM

gcc/gfortran

Other projects used

- **g95-xml** – the XML markup for Fortran 95 source code based on g95 compiler (by Philippe Marguinaud). Used as input for code split transformations
- **LLVM Dragonegg** – bridge to utilize GCC as frontend to LLVM ⇒ compile Fortran code (by Duncan Sands et al)
- **LLVM C backend** – C code generator out of LLVM IR (by Chris Lattner, Duncan Sands et al)

KernelGen preview release

Project source code, docs and binaries at HPCForge:

http://hpcforge.org/projects/kernelgen/

Binaries for 64-bit Fedora 15:

kernelgen-0.1-cuda.x86_64.rpm
kernelgen-0.1-opencl.x86_64.rpm

Documentation on wiki:

Running the public test suite
Compiling (for developers)

Collaboration

We provide:

- Source code and binaries
- User support, updates and bug fixes

We need:

- Users feedback, testing and filing bugs
- Access to actual benchmarks (our COSMO is v4.13)
- Developers are welcome, especially skilled in LLVM and/or models

Thank you! ☺ Questions?