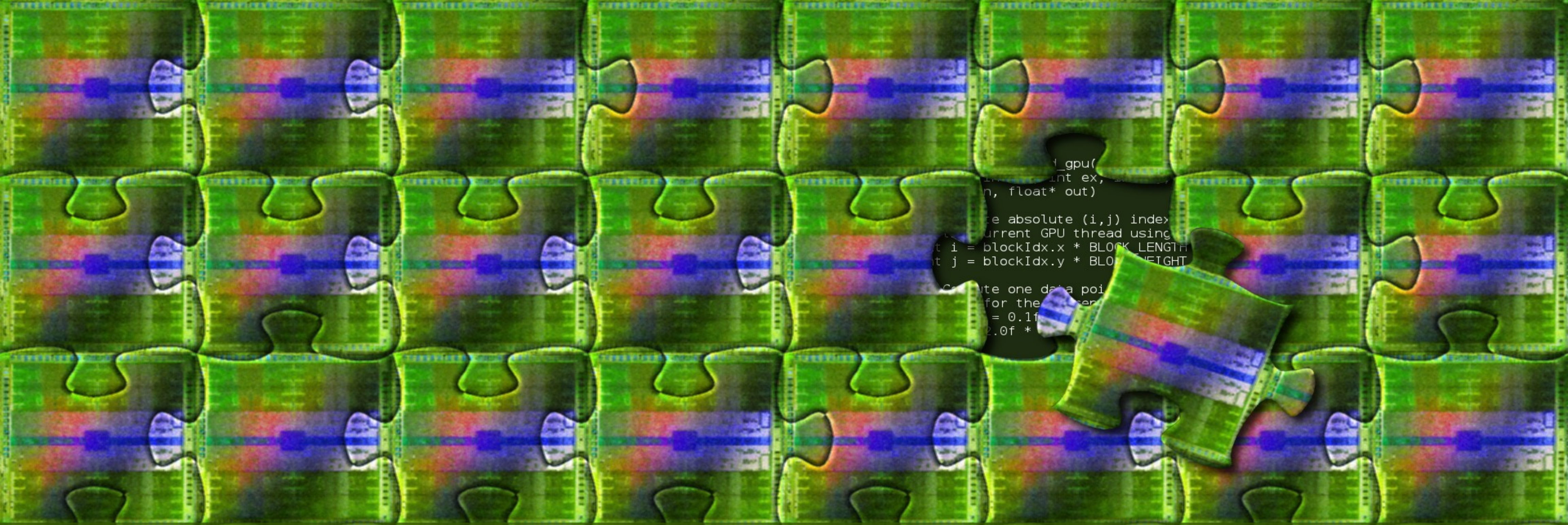


KernelGen naïve GPU kernels generation from Fortran source code

Dmitry Mikushin

Contents

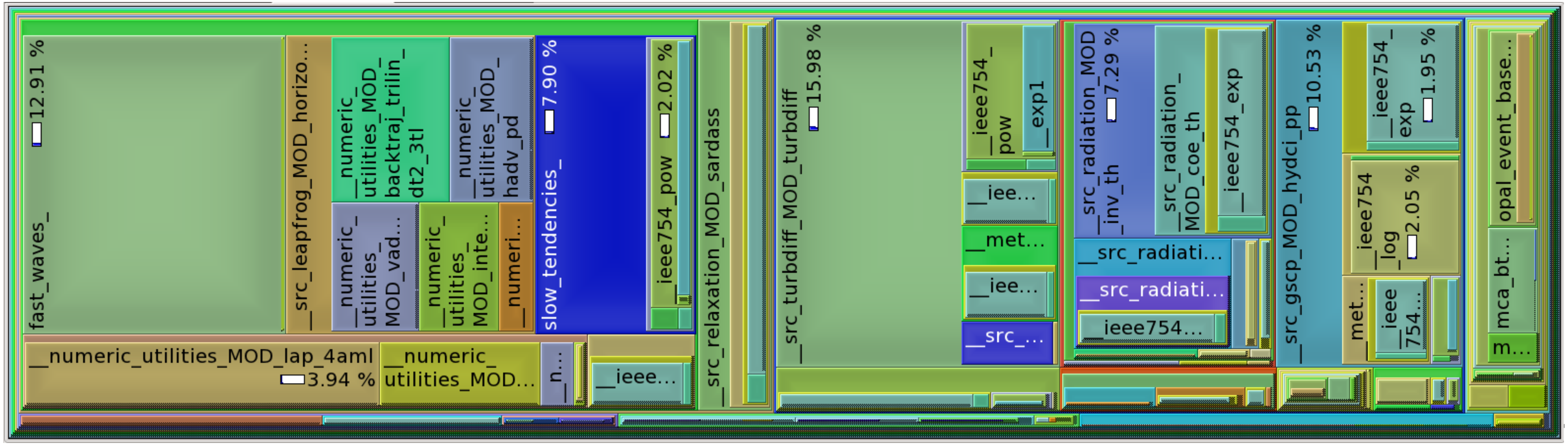
- Motivation, target, analysis
- Assembling our own toolchain
- Toolchain usecase: axpy example
- Development schedule

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

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

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

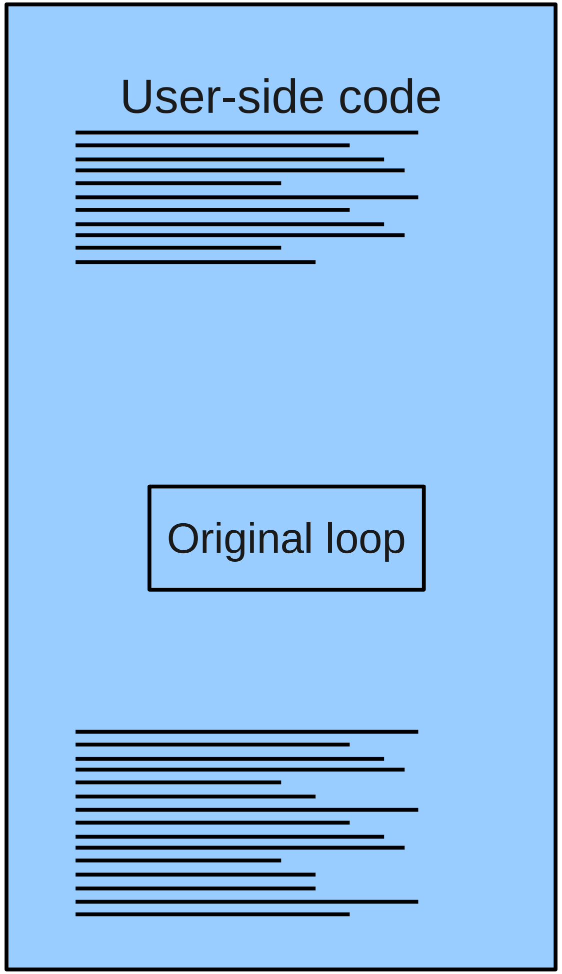
- Can we build our own toolchain with dependencies analysis instead of directives?

2. Assembling our own toolchain

Ingredients

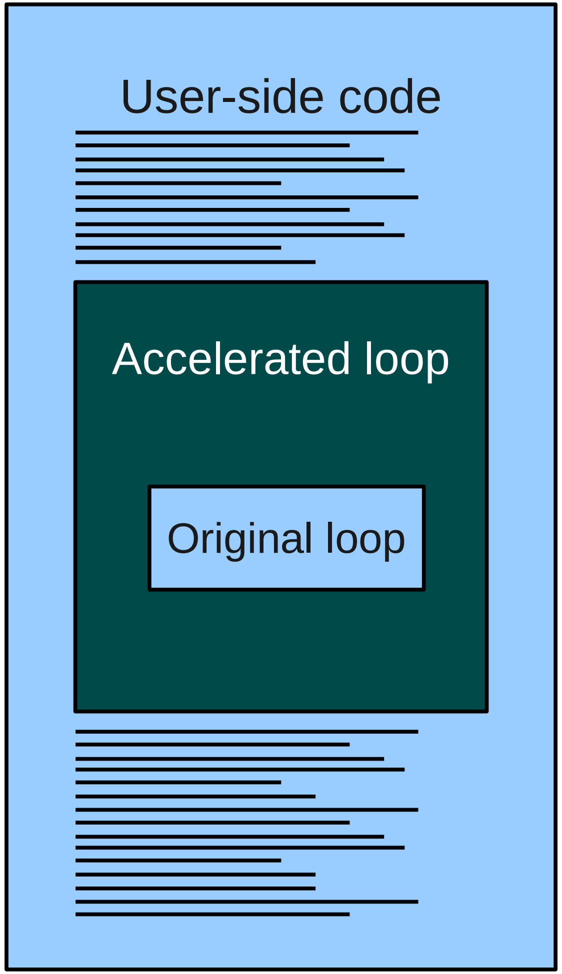
- **Compiler** – split original code into host and device parts and compile them into single object
  - Code splitter (source-to-source preprocessor)
  - Target device code generator
- **Runtime library** – implementation of specific internal functions used in generated code
  - Data management
  - Kernel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

Runtime workflow



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

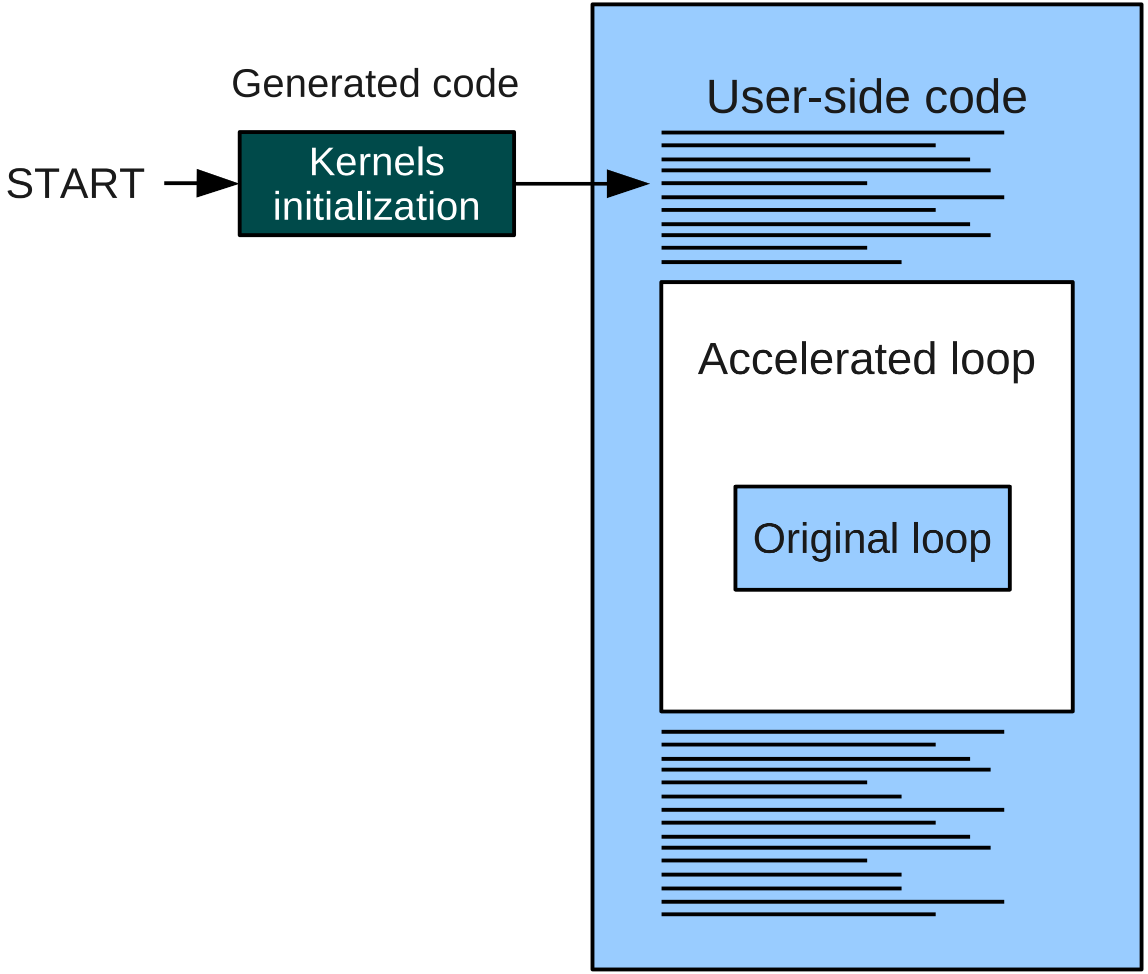
Runtime workflow



Equivalent device code is generated for suitable loops.

(see “Code generation workflow” for details)

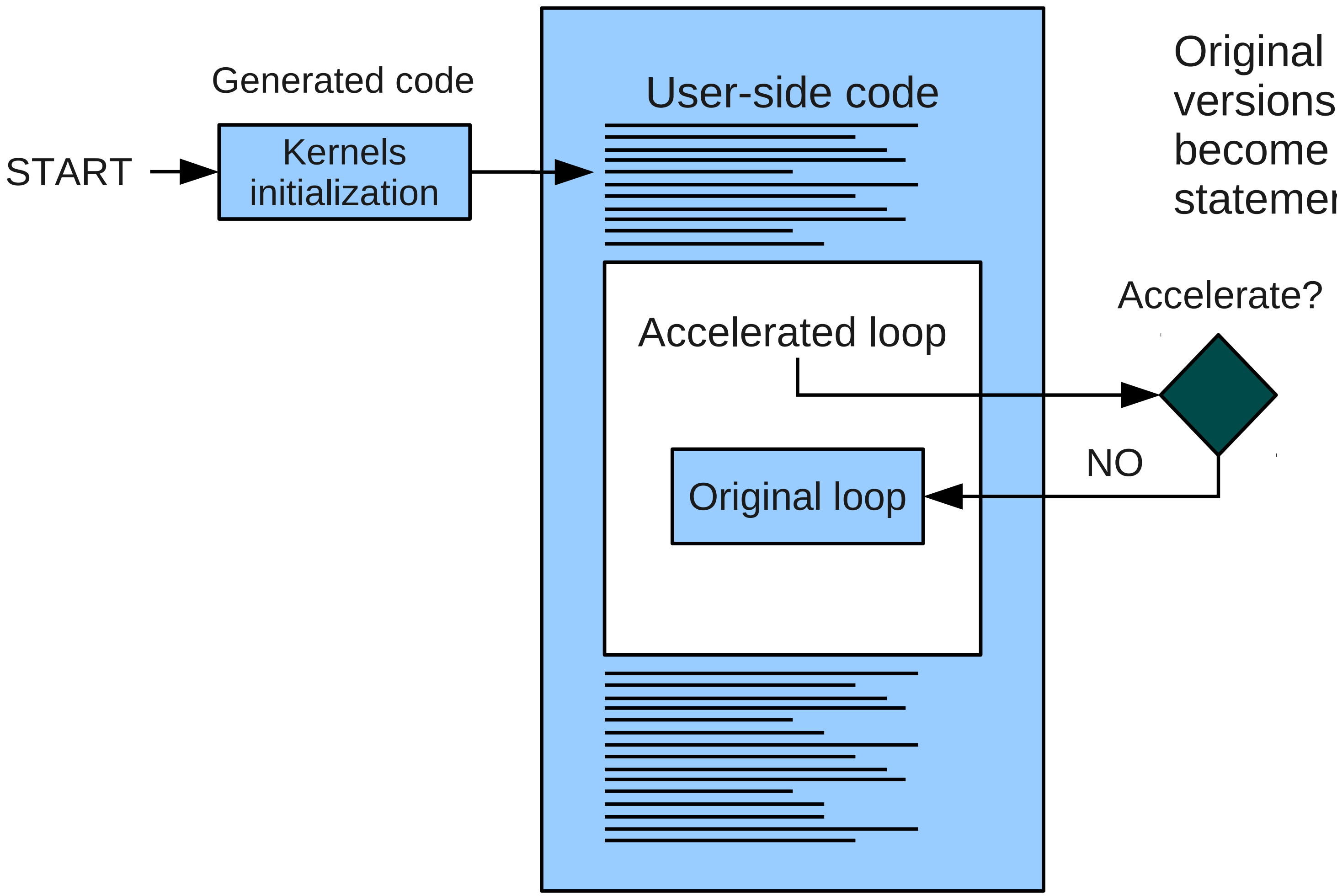
Runtime workflow



Equivalent device code is generated for suitable loops.

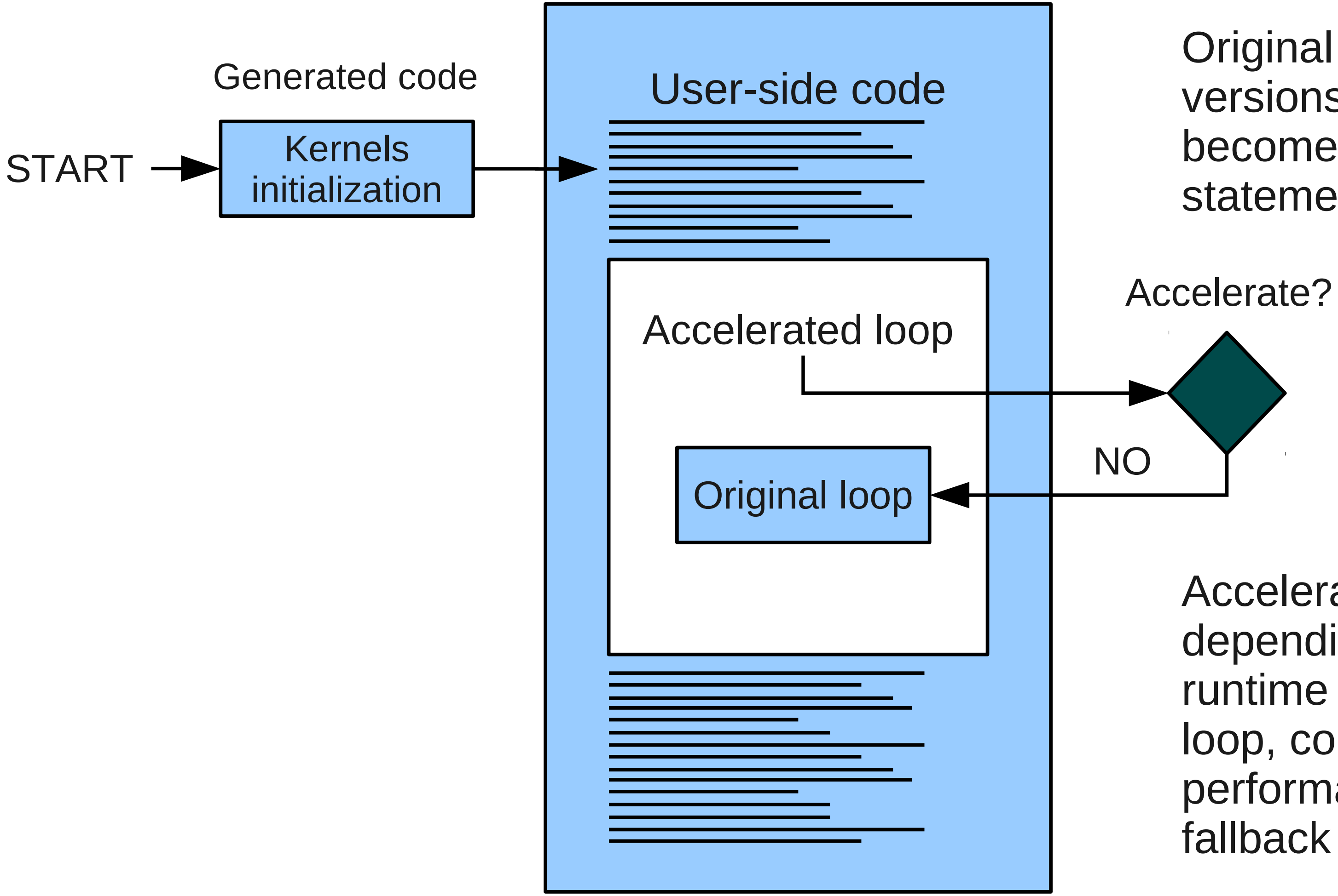
Additionally global constructors are generated to initialize configuration structures (with status, profiling, permanent dependencies, etc.) for each kernel.

Runtime workflow



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

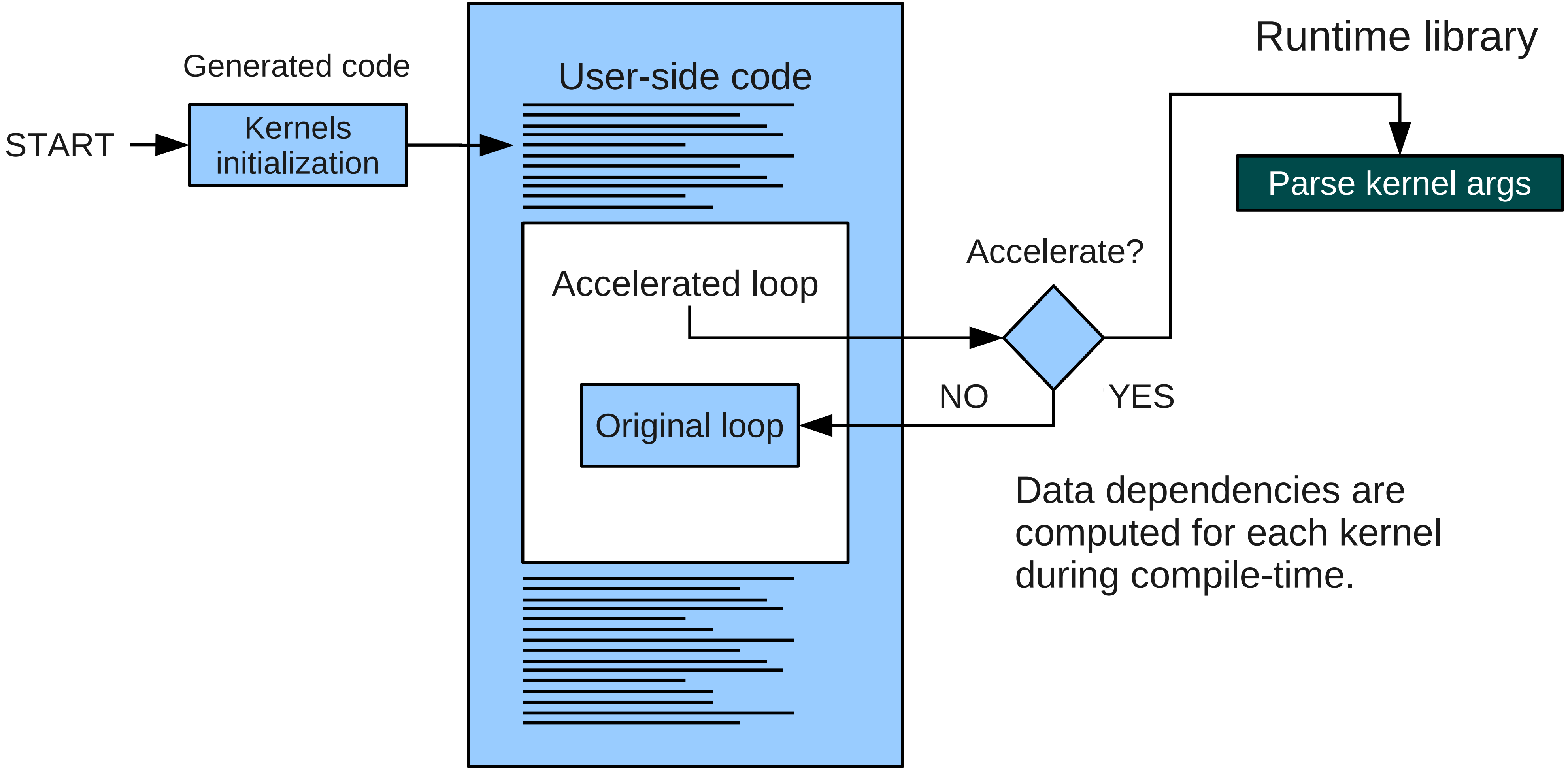
Runtime workflow



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

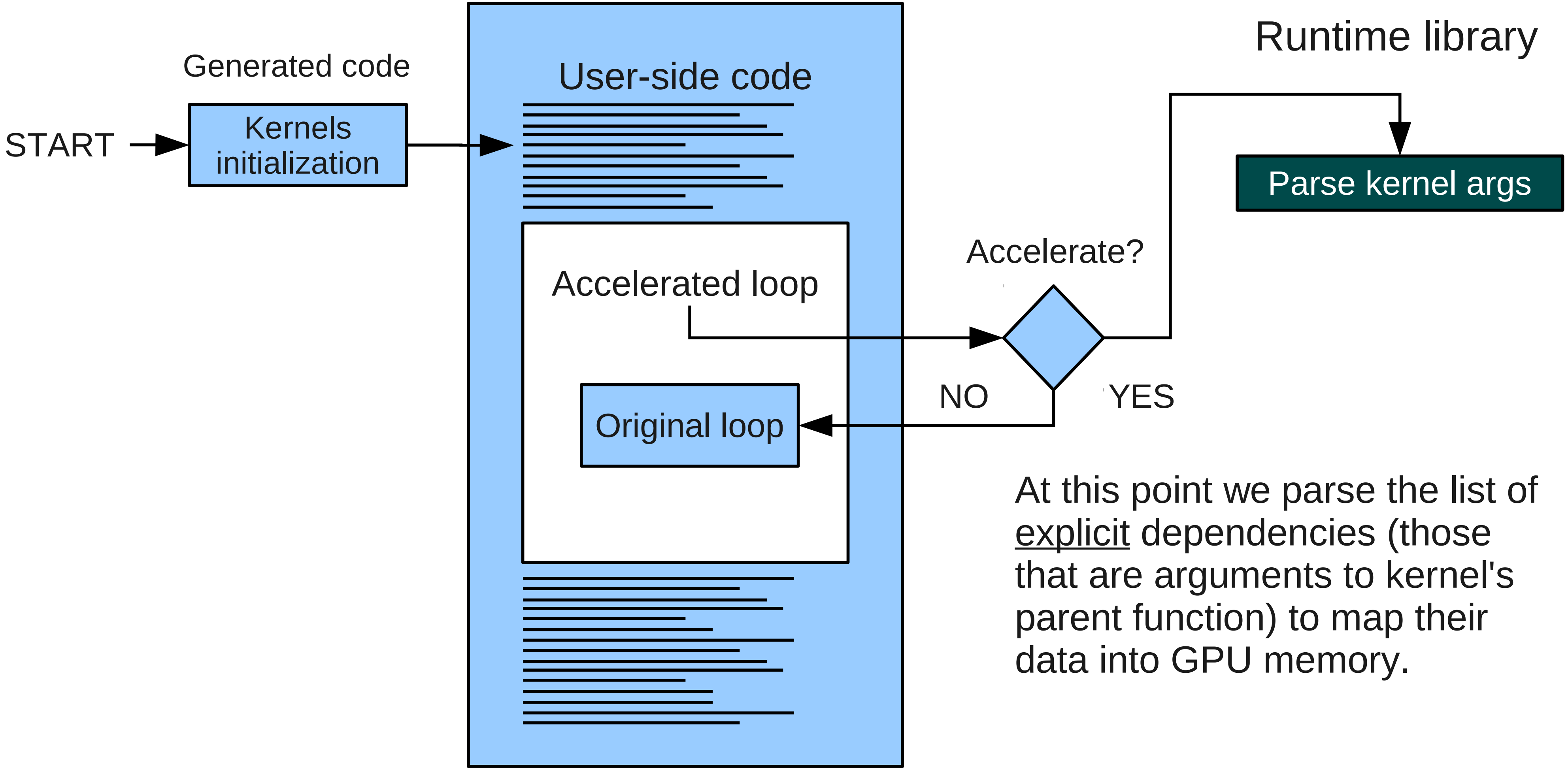
Accelerated loop is selected, depending on the number of runtime properties (not failing loop, correct results, better performance, etc.); otherwise – fallback to original version.

Runtime workflow



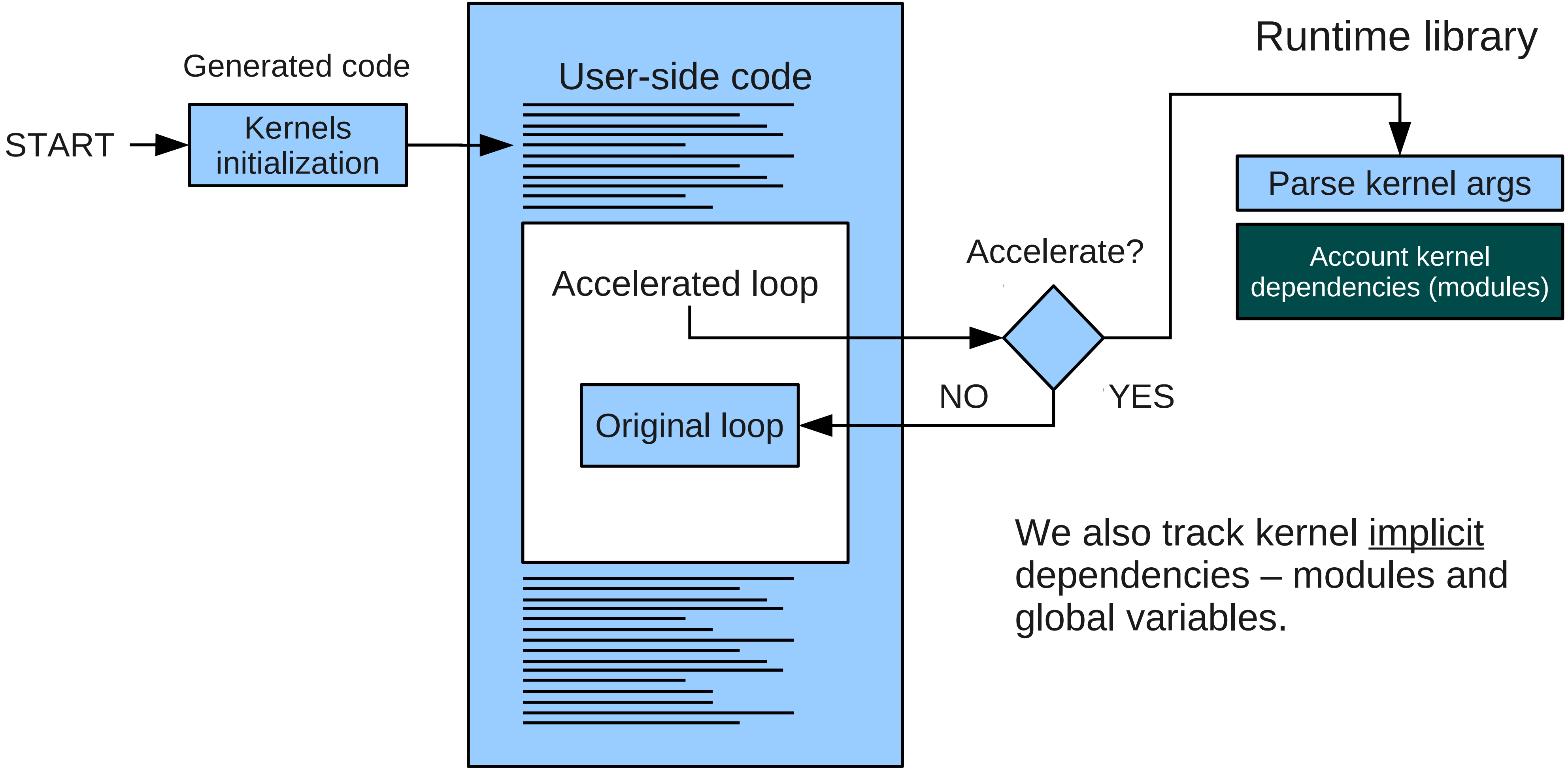
Data dependencies are computed for each kernel during compile-time.

Runtime workflow



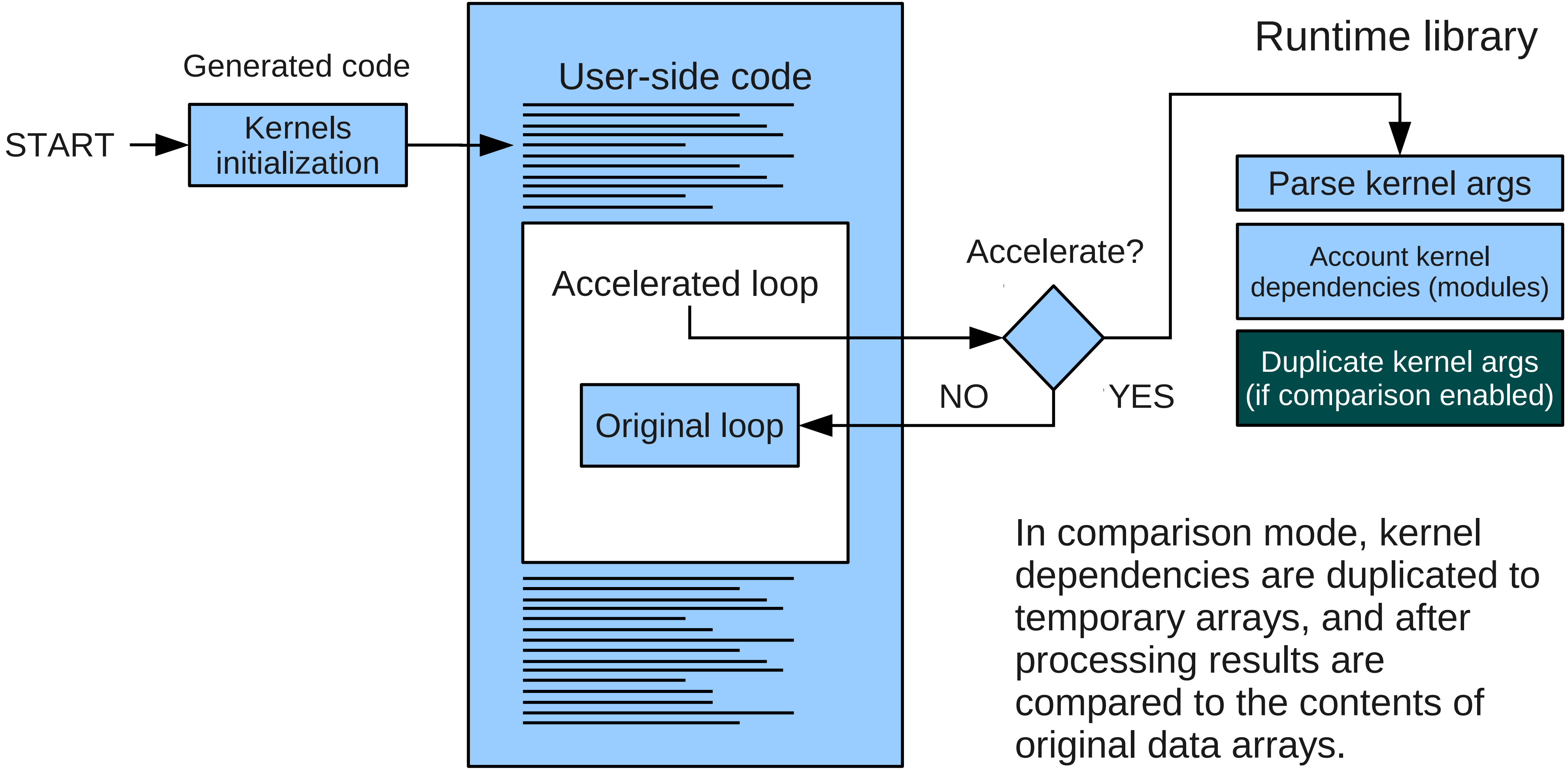
At this point we parse the list of <u>explicit</u> dependencies (those that are arguments to kernel's parent function) to map their data into GPU memory.

Runtime workflow



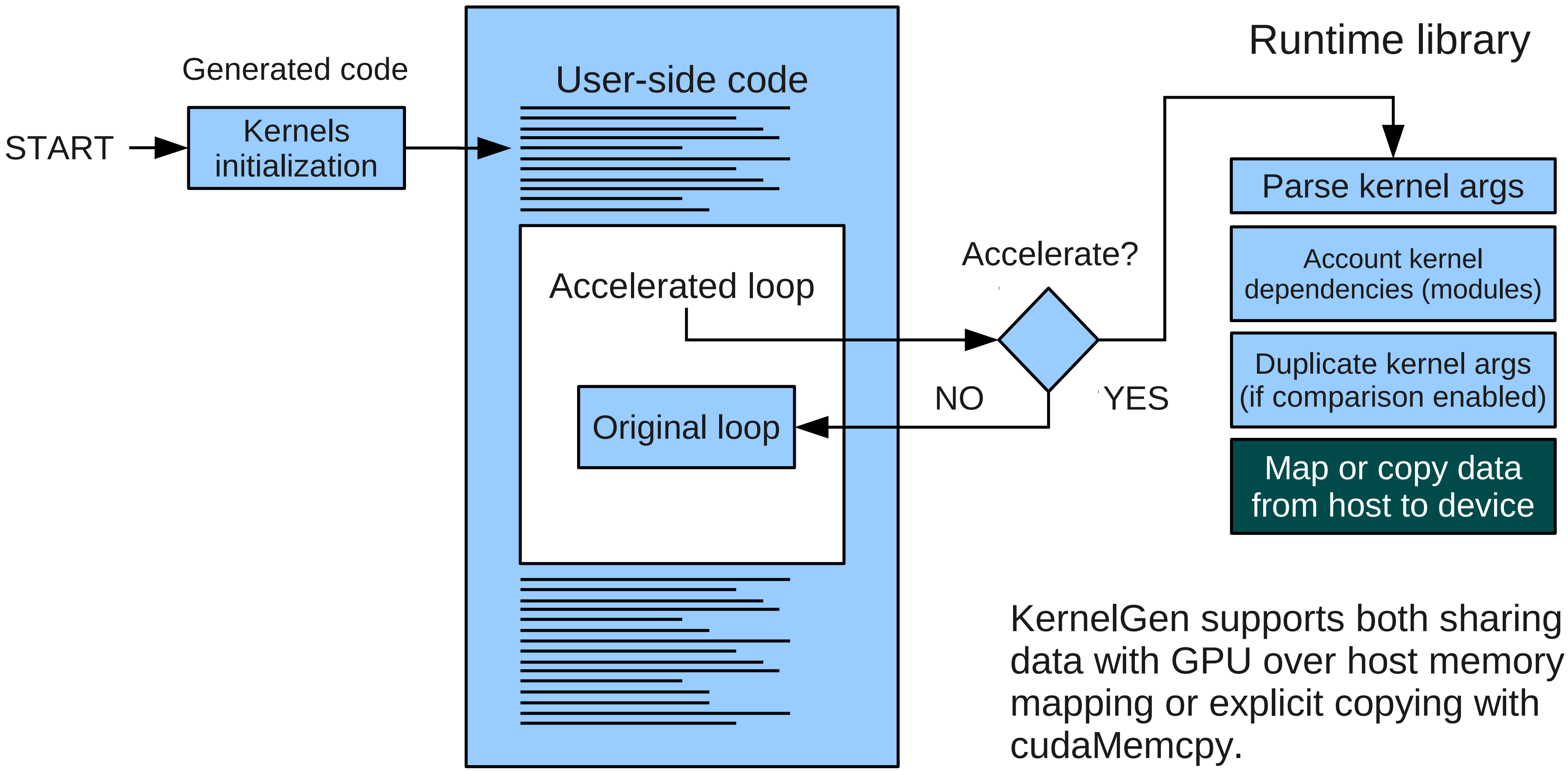
We also track kernel <u>implicit</u> dependencies – modules and global variables.

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.

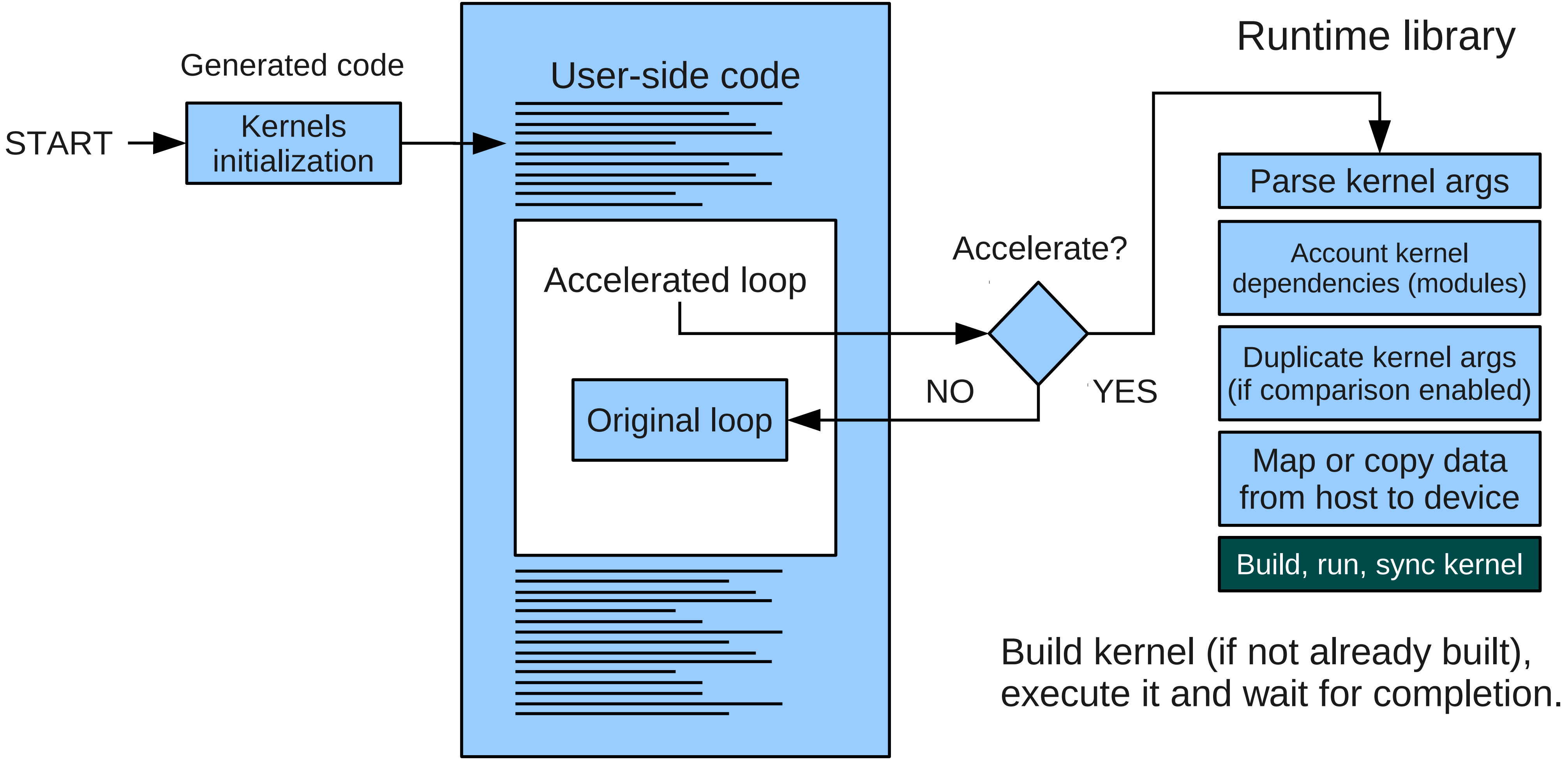
Runtime workflow



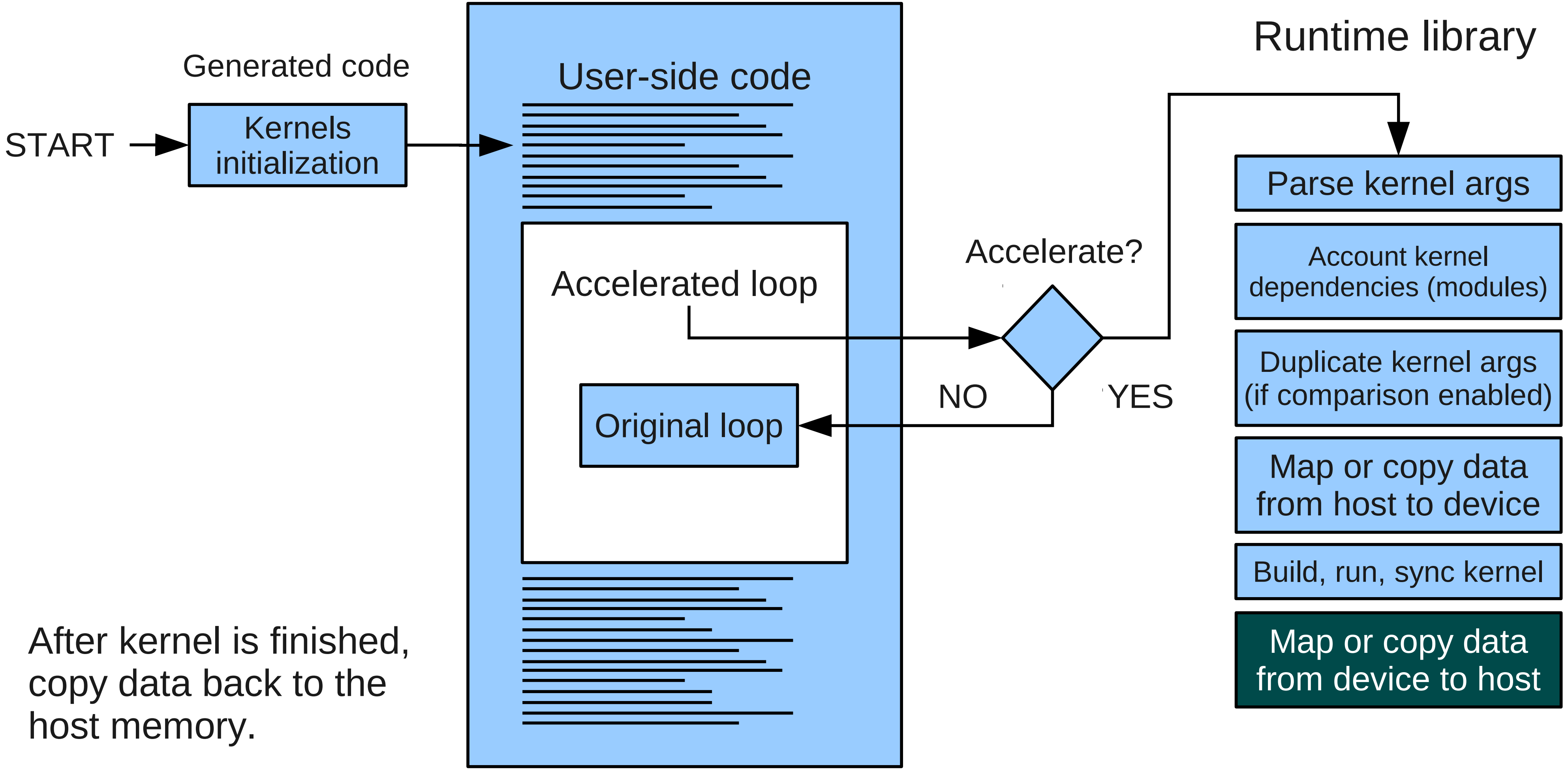
KernelGen supports both sharing data with GPU over host memory mapping or explicit copying with cudaMemcpy.

2011

Runtime workflow

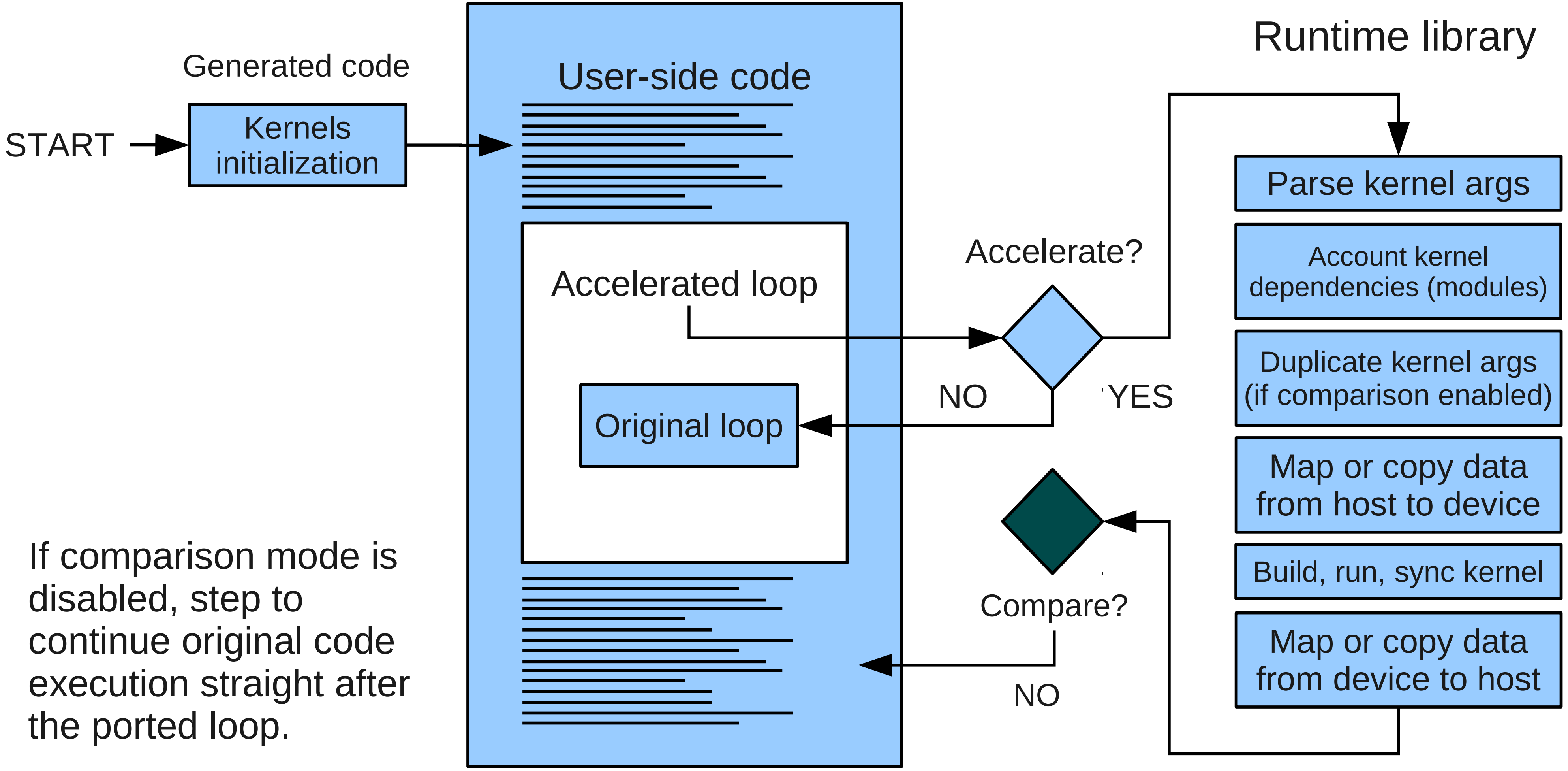


Runtime workflow



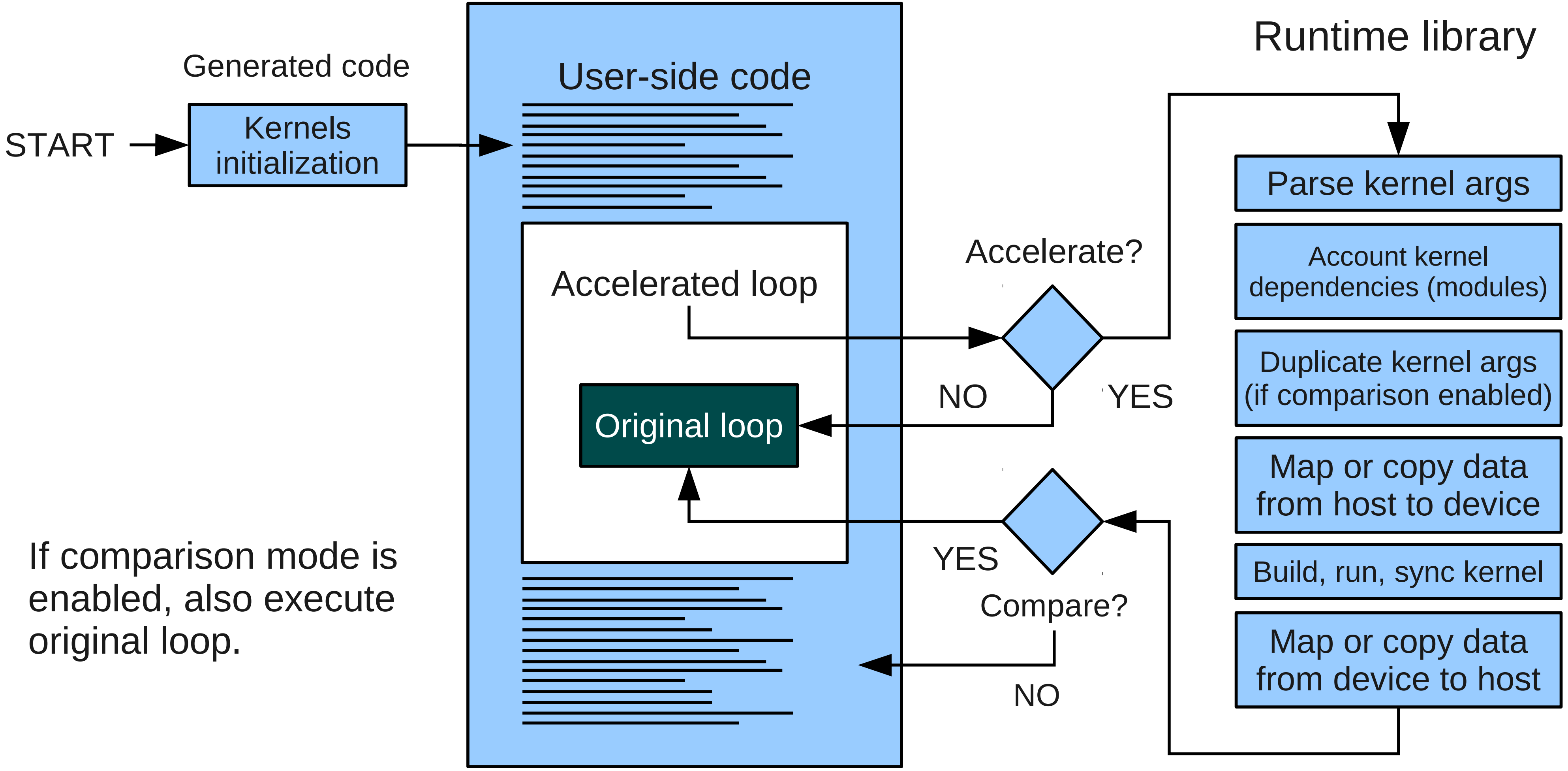
After kernel is finished, copy data back to the host memory.

Runtime workflow



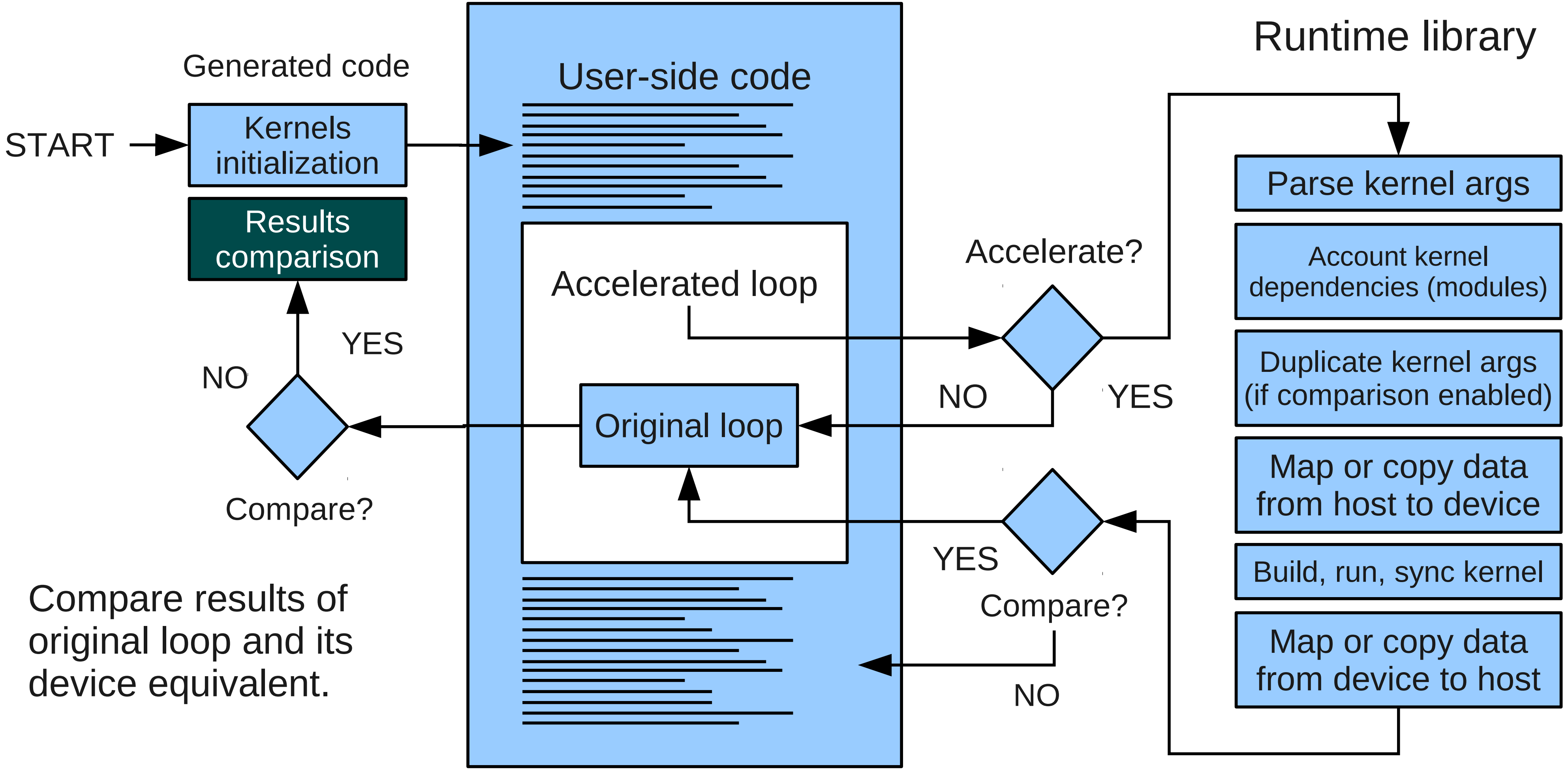
If comparison mode is disabled, step to continue original code execution straight after the ported loop.

Runtime workflow



If comparison mode is enabled, also execute original loop.

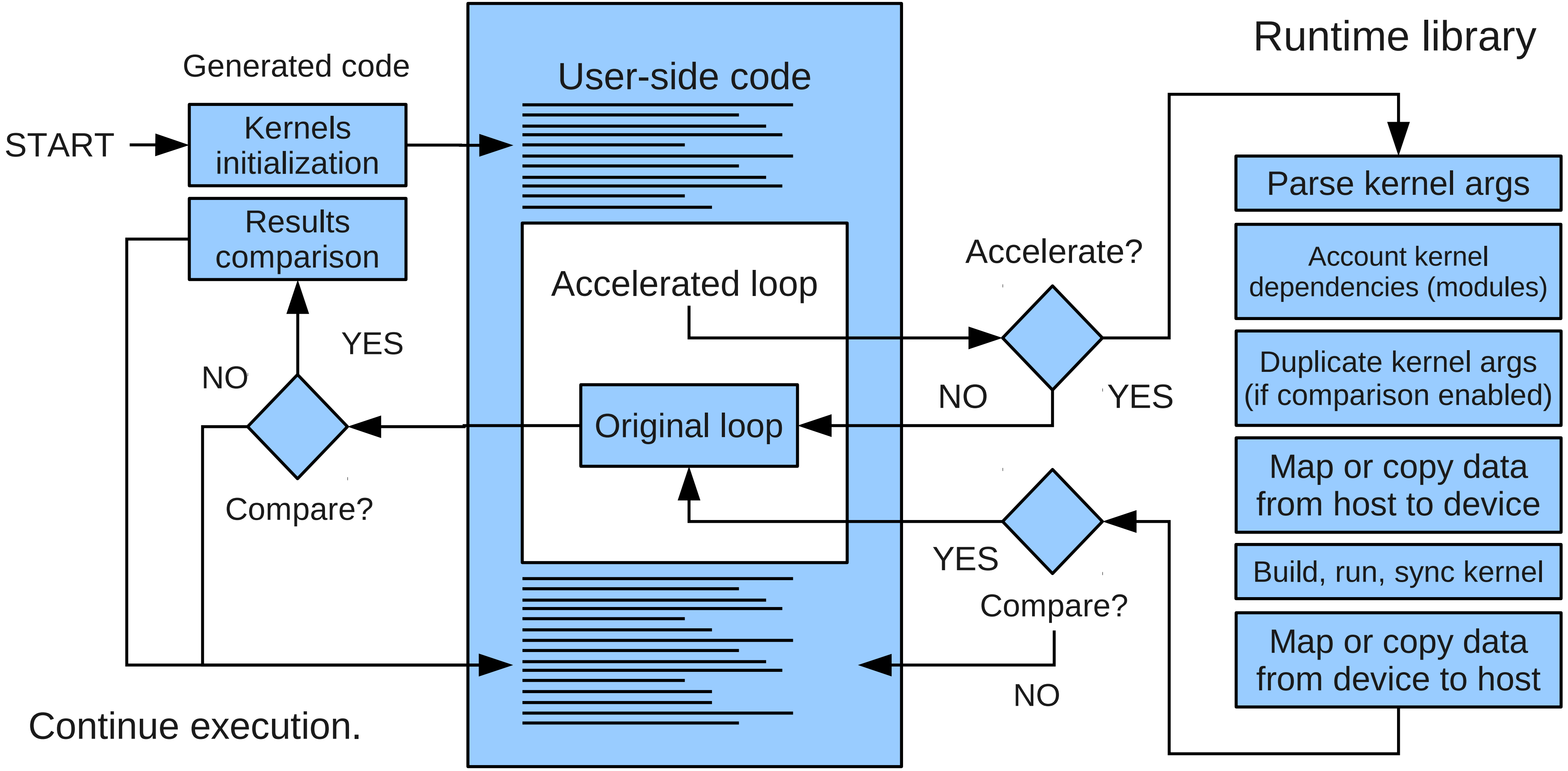
Runtime workflow



Compare results of original loop and its device equivalent.

2011

Runtime workflow



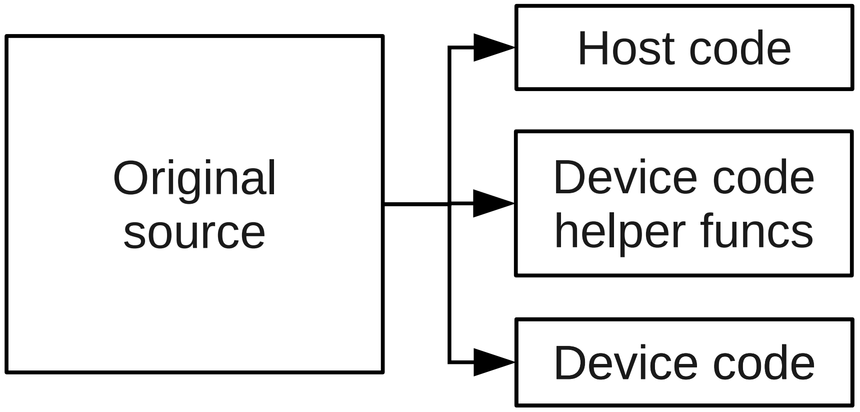
2011

Code generation workflow

Two parts of code generation process:

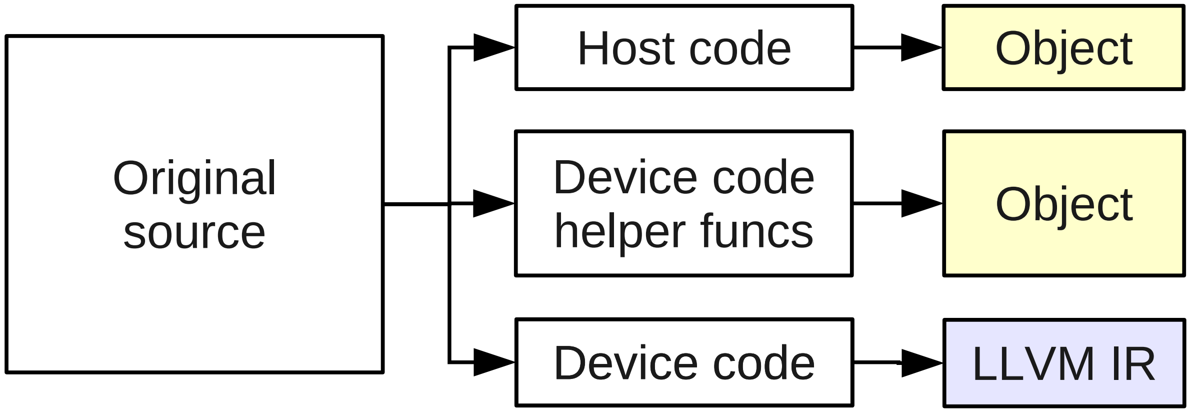
- **Compile time** – generate kernels strictly corresponding to original host loops
- **Runtime** – generate kernels, using additional info: inline external functions, optimize compute grid, etc.

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.

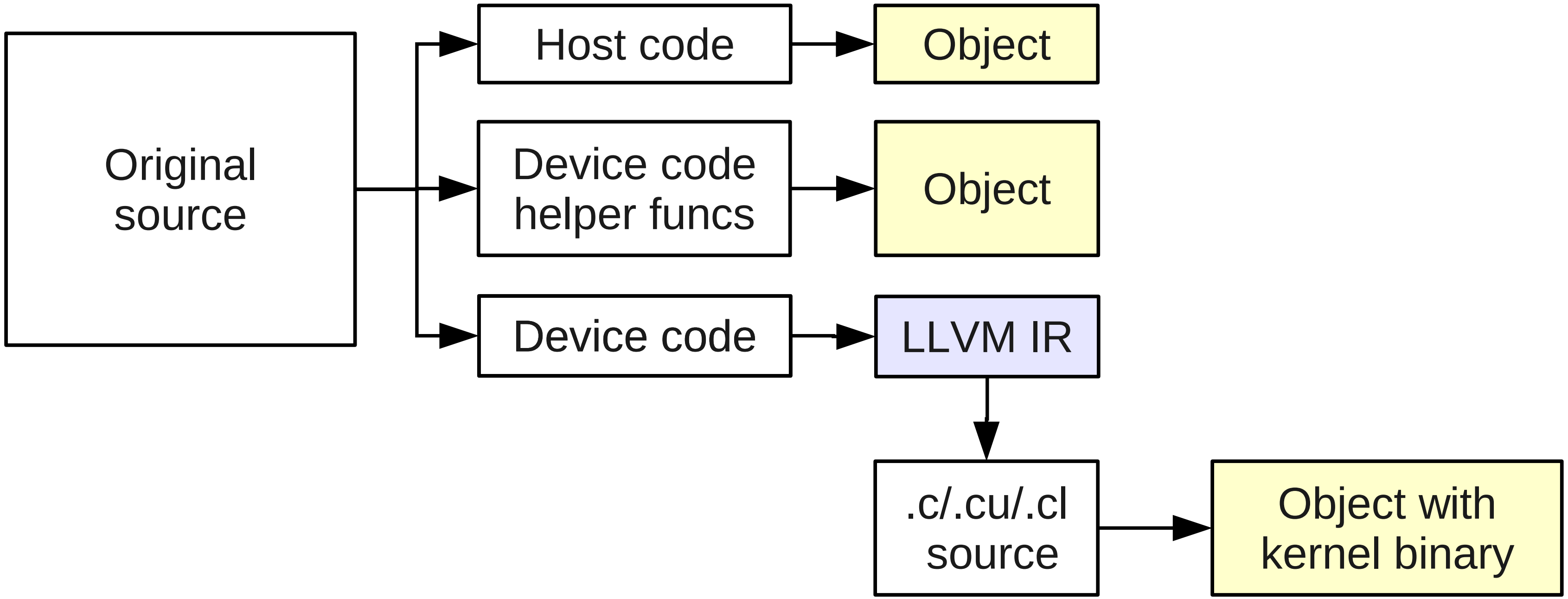
Code generation workflow (compile-time part)



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

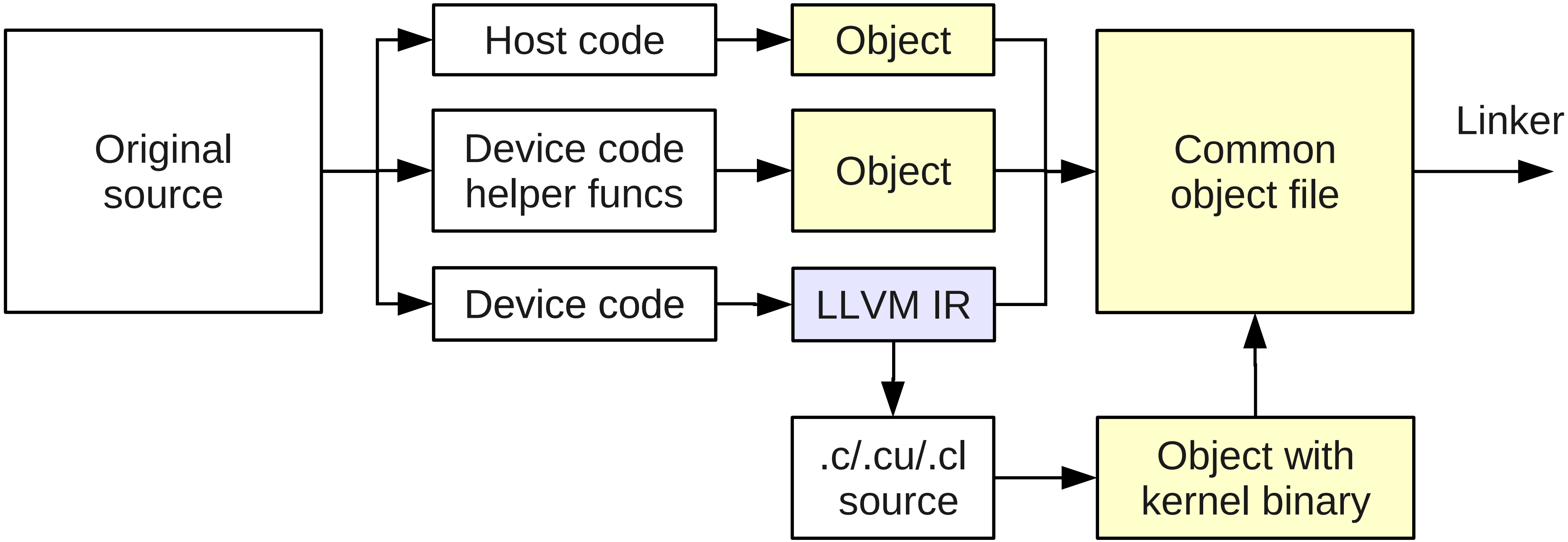
Device code is compiled into Low-Level Virtual Machine Intermediate representation (LLVM IR).

Code generation workflow (compile-time part)



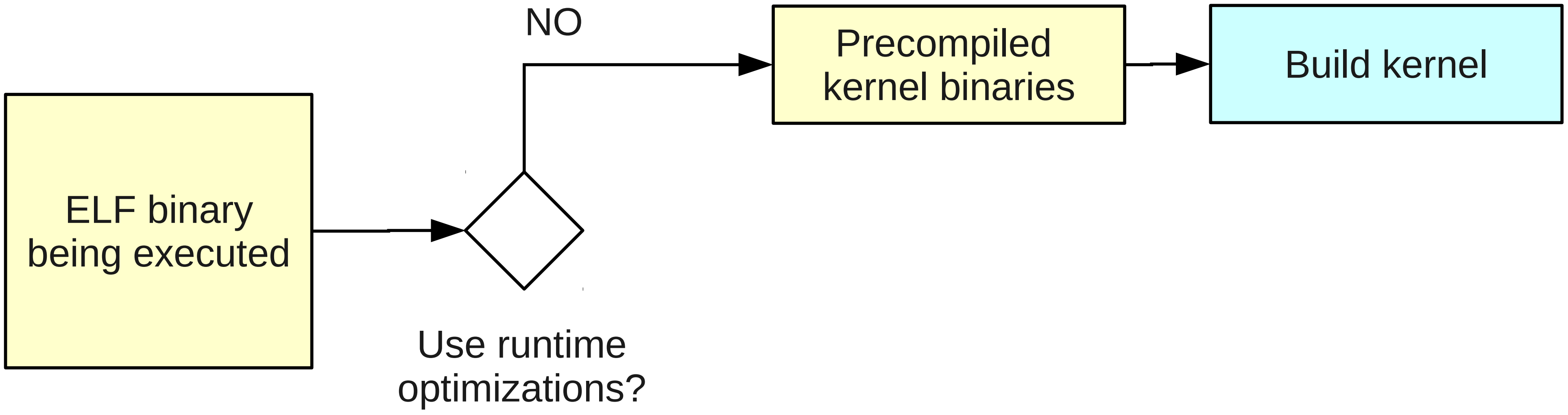
Code from LLVM IR is translated into C, CUDA or OpenCL using modified LLVM C Backend and compiled using the corresponding device compiler.

Code generation workflow (compile-time part)



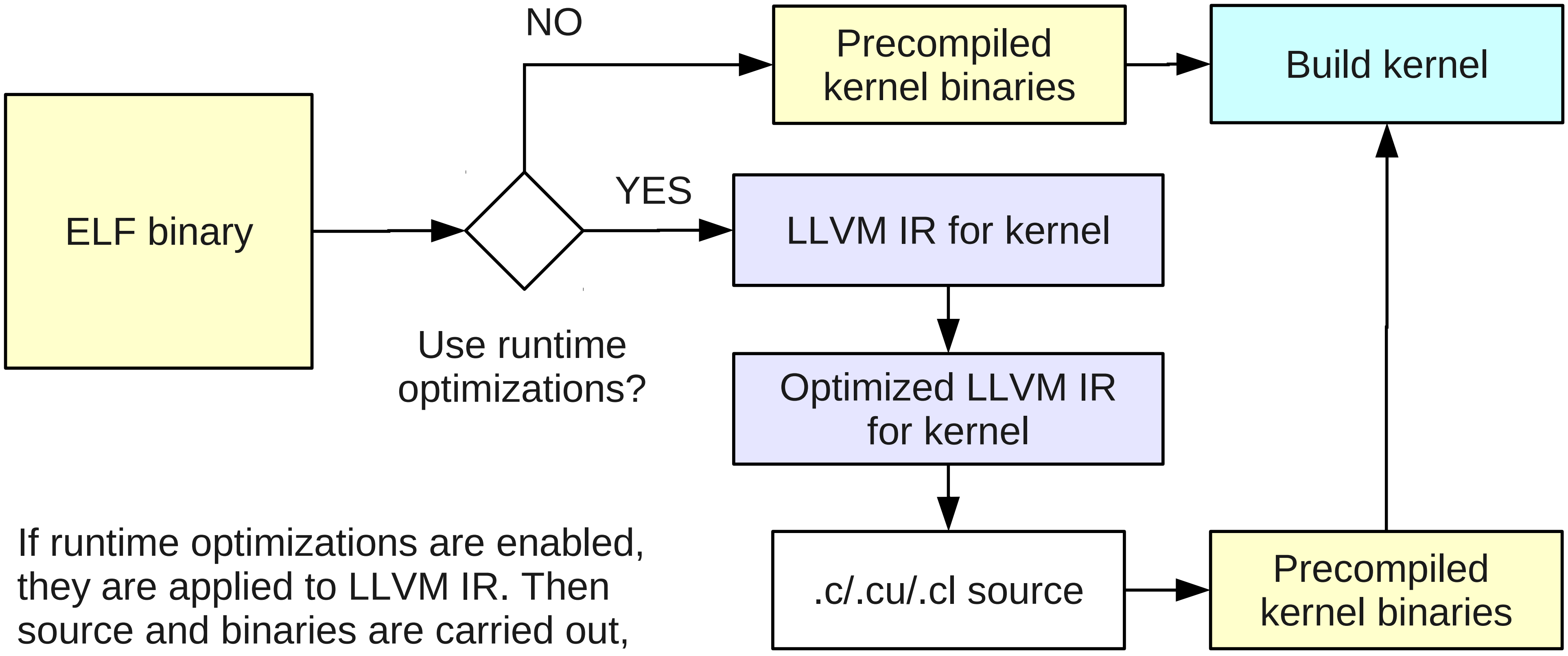
Finally, objects for all parts of the code are merged into single object to conserve "1 source → 1 object" layout. LLVM IR is also embedded into resulting object.

Code generation workflow (runtime part)



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

Code generation workflow (runtime part)



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. Development strategy

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

4. Toolchain internals

Example: sincos

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

```
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)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$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)

```
!$KERNELGEN SELECT sincos_loop_1_kernelgen
if (sincos_loop_1_kernelgen_config%runmode .ne. kernelgen_runmode_host) then
!$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
```

Execute original loop

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

!$acc end region

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

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)

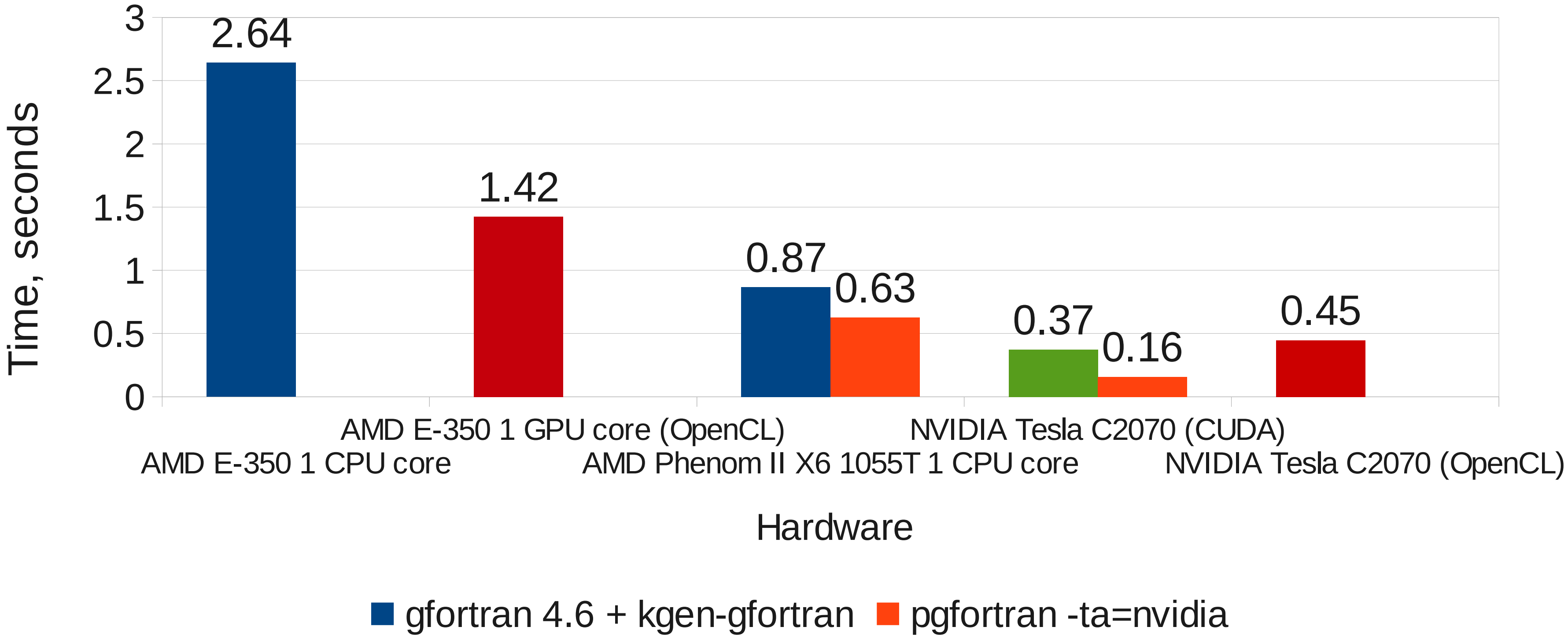
```
[marcusmae@T61p axpy]$ axpy_1=1 ./axpy 1000 0.001
arg 0 maps memory segment [140735344500736 .. 140735344508928] to
[1052672 .. 1060864]
arg 1 maps memory segment [28172288 .. 28184576] to [1060864 .. 1073152]
arg 2 reuses mapping created by arg 0
arg 3 reuses mapping created by arg 1
 Value of i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

5. Testing unoptimized generator

2011

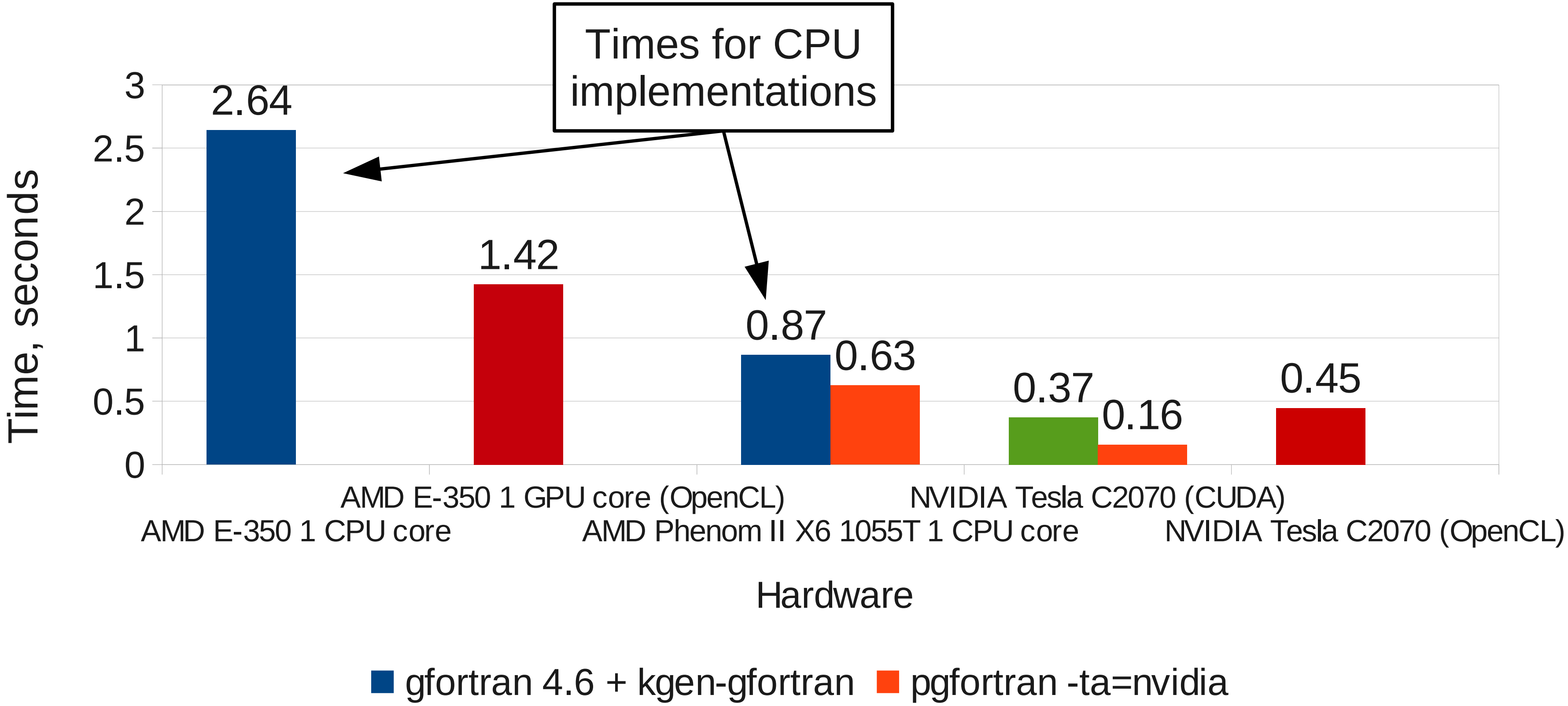
Example: sincos – performance

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



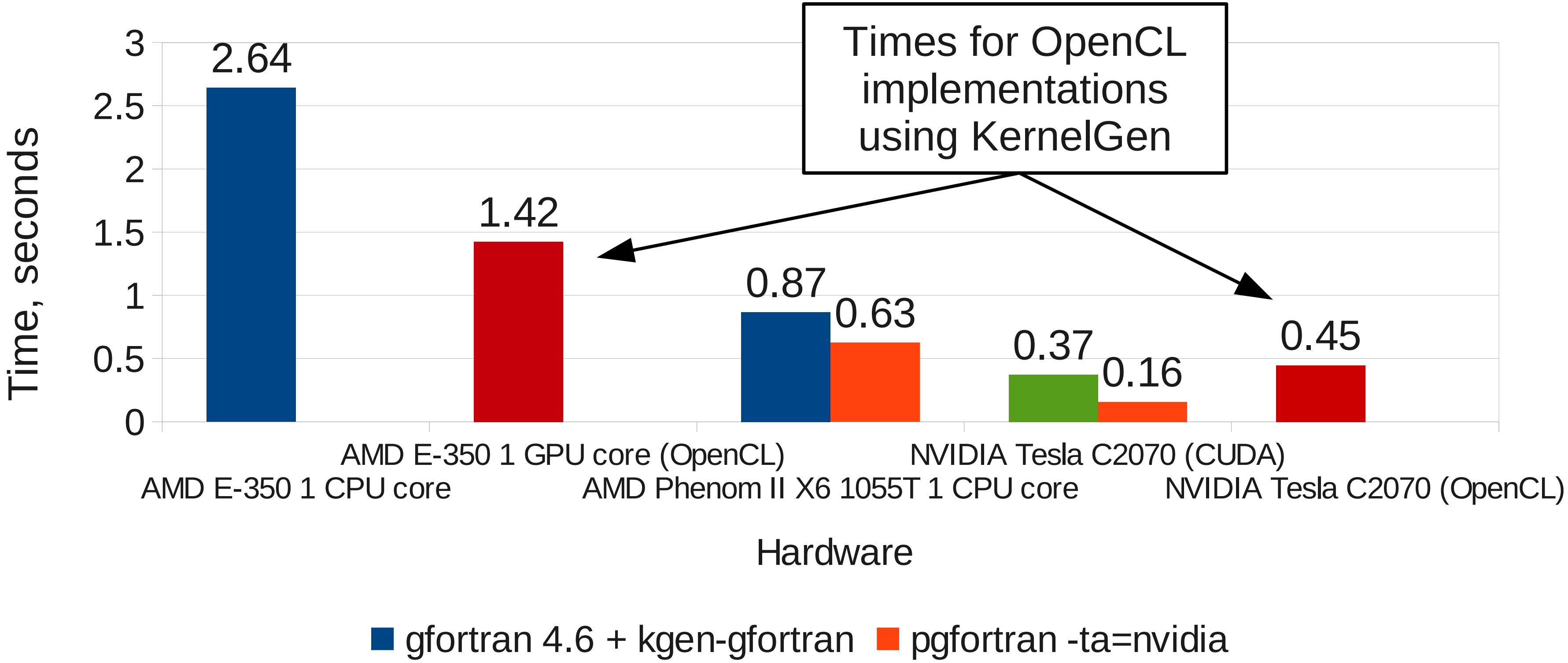
Example: sincos – performance

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



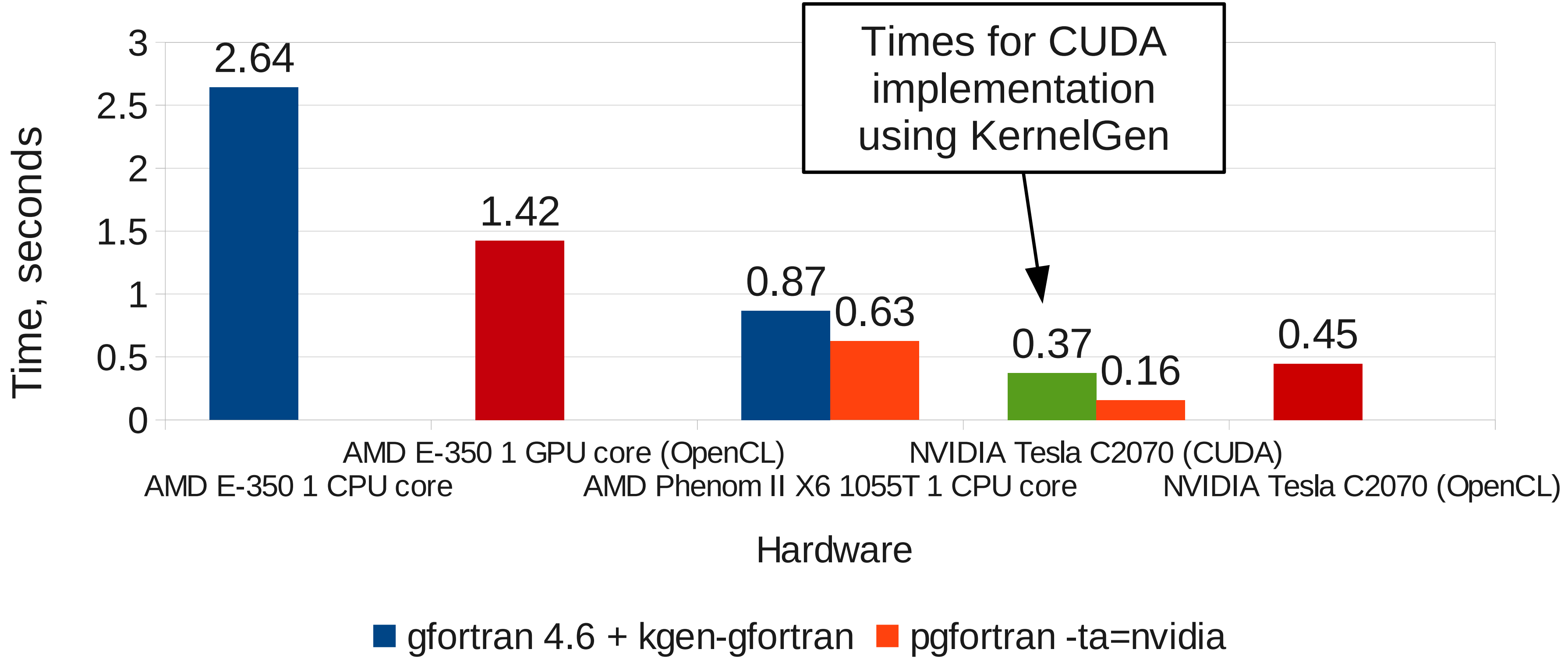
Example: sincos – performance

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



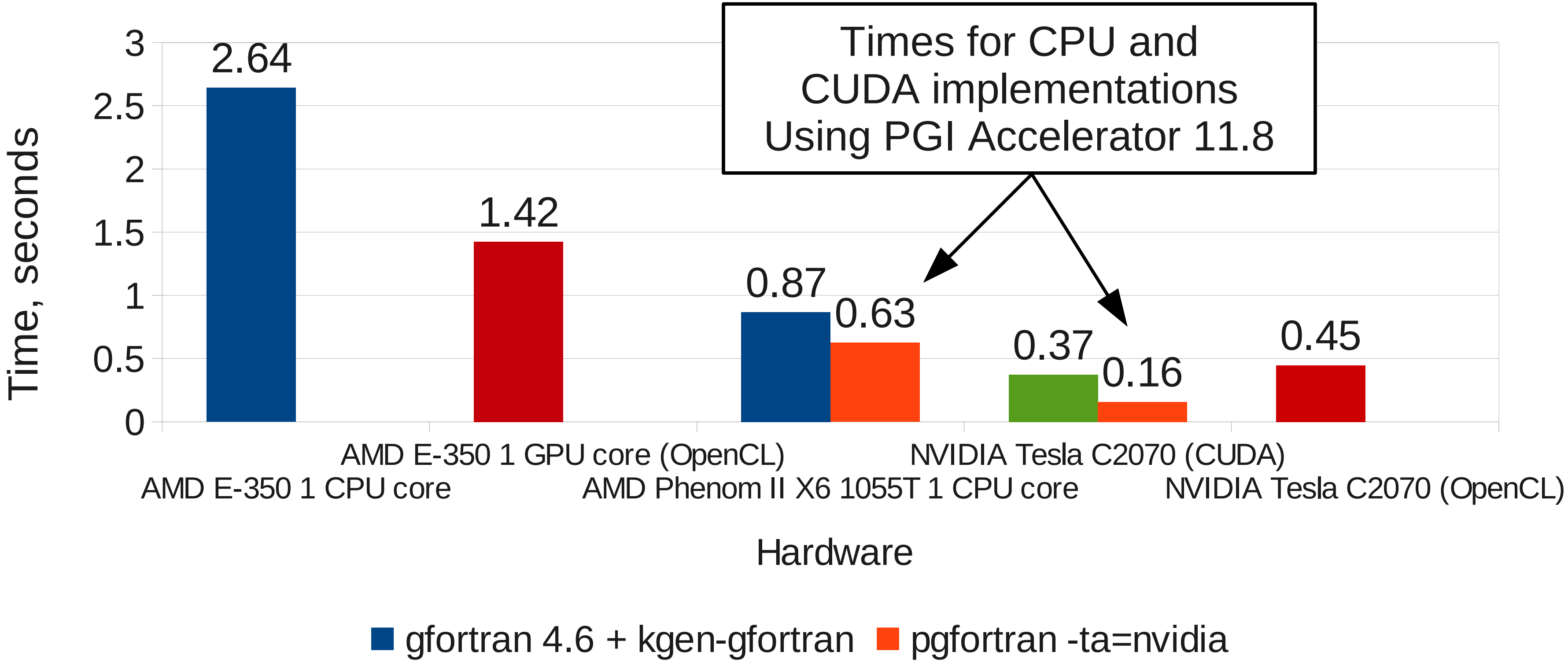
Example: sincos – performance

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

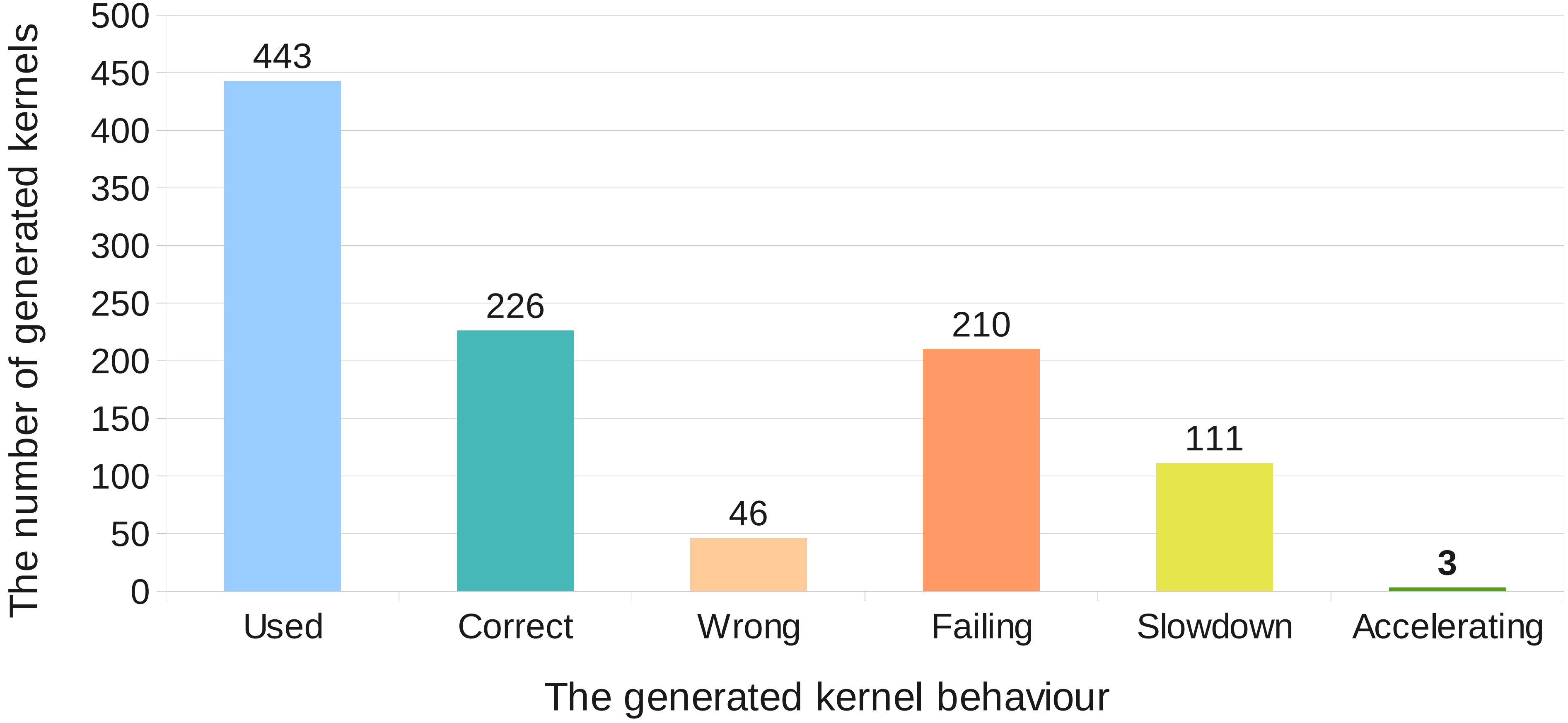


Example: sincos – performance

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)

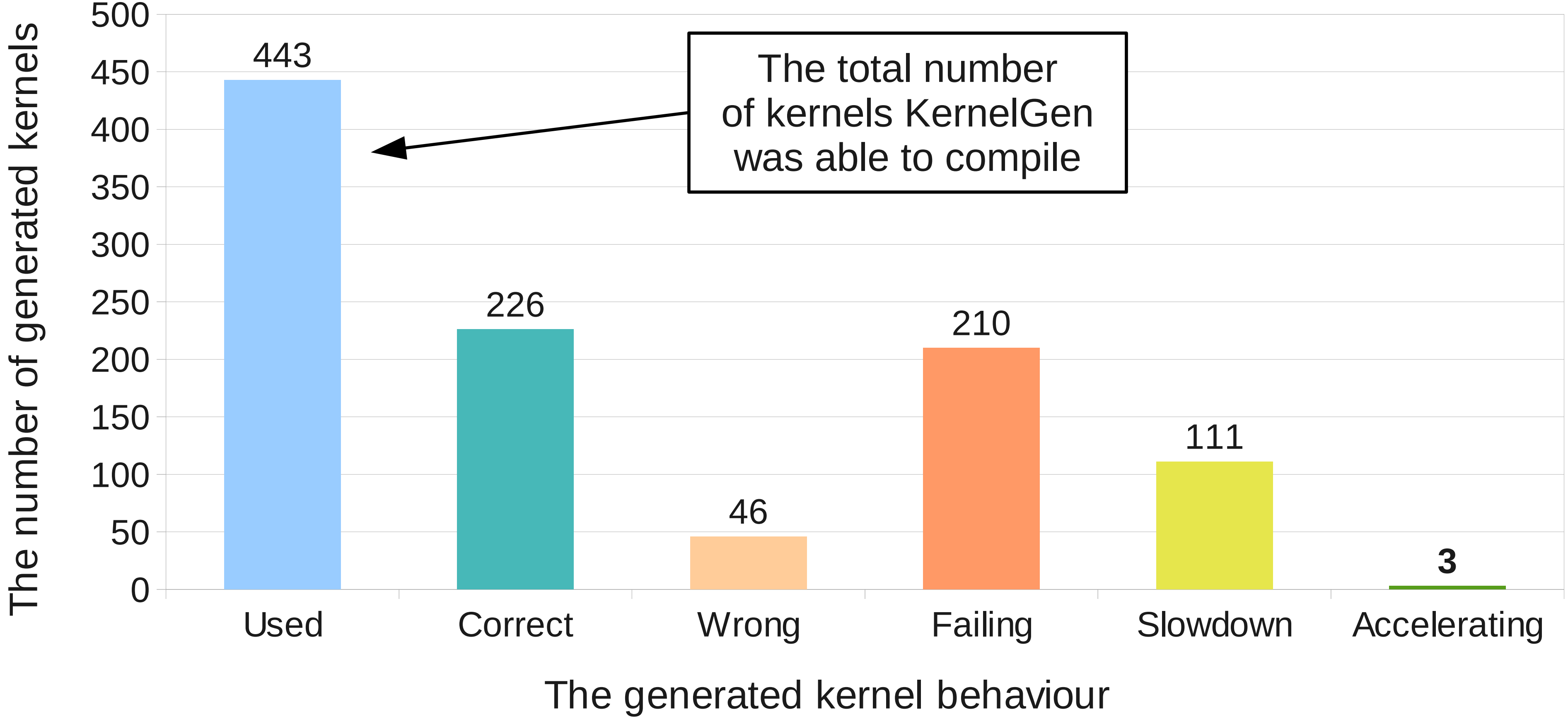


COSMO – coverage

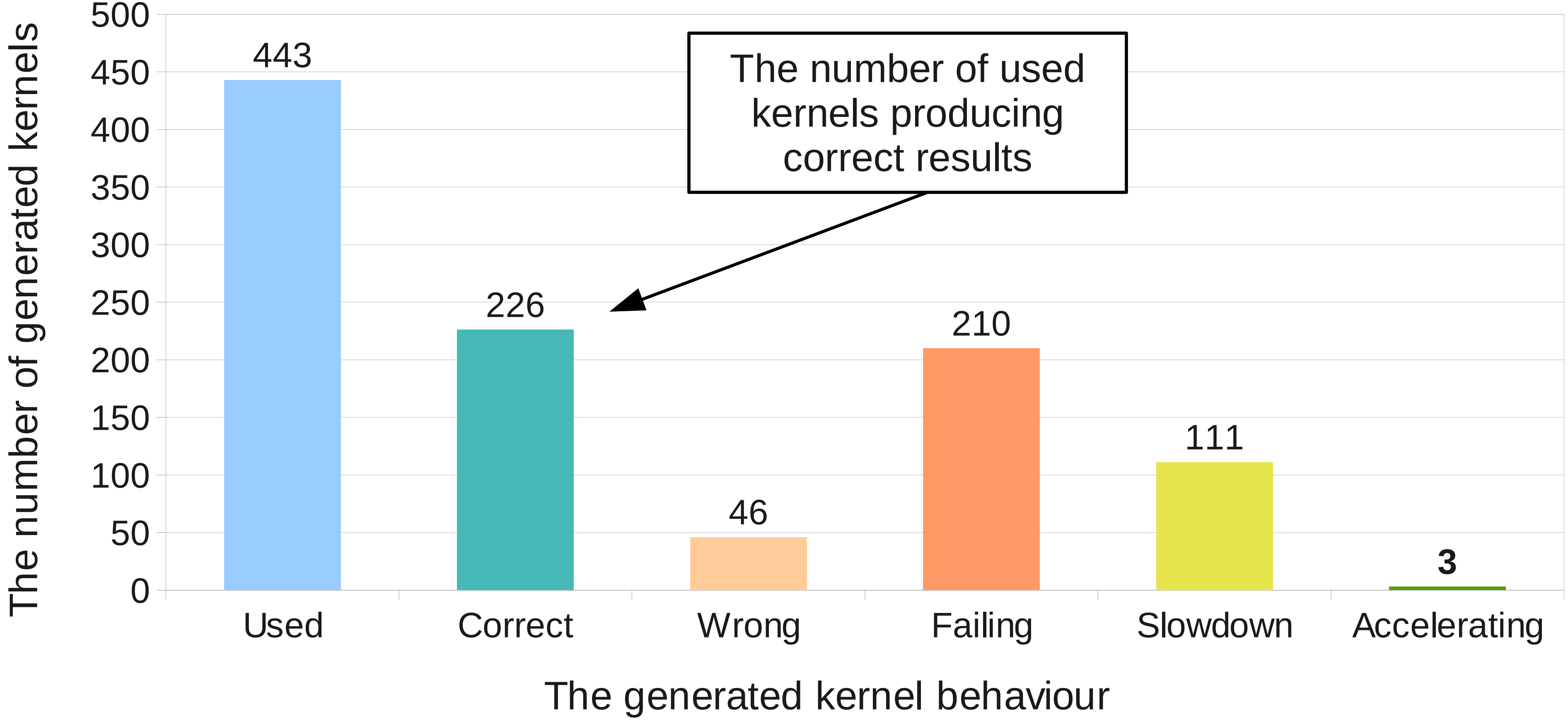


2011

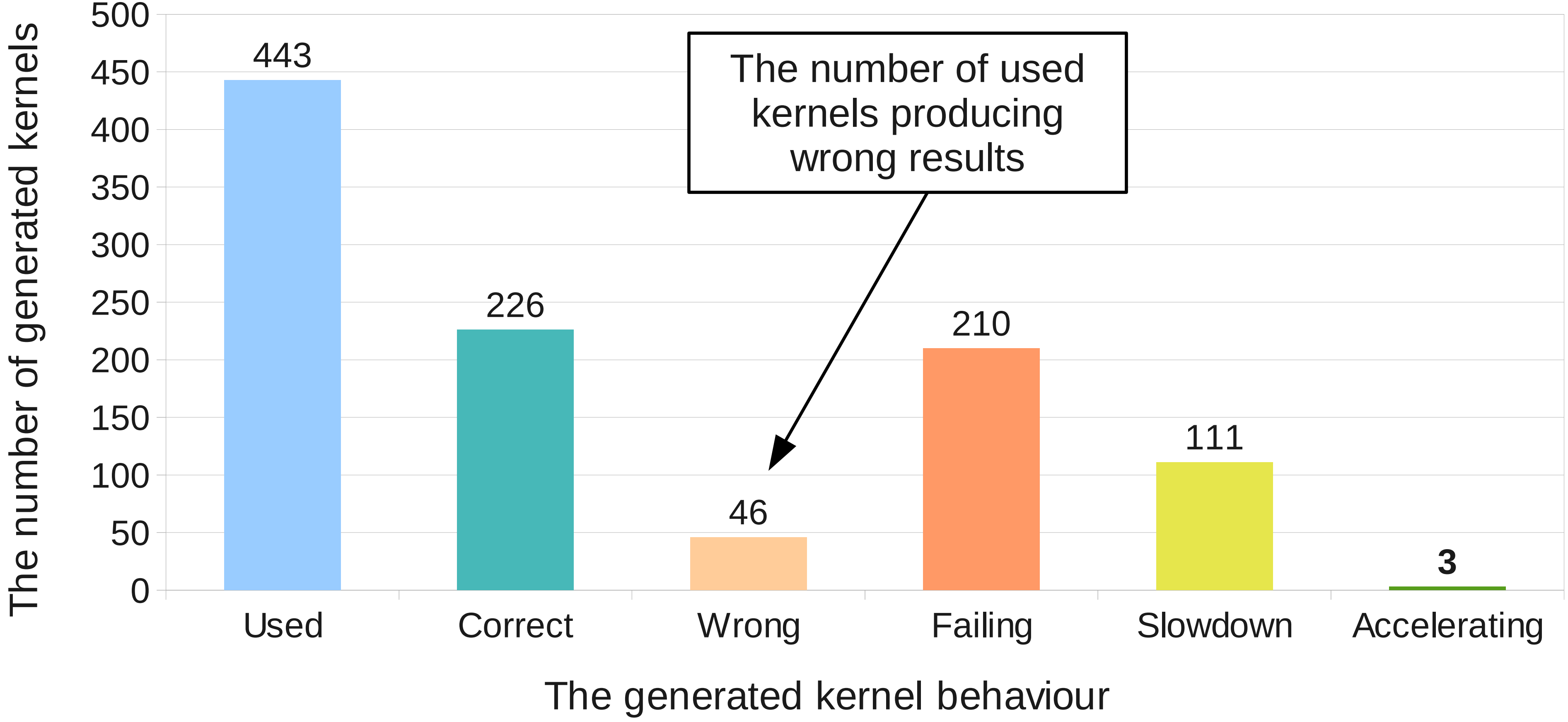
COSMO – coverage



COSMO – coverage

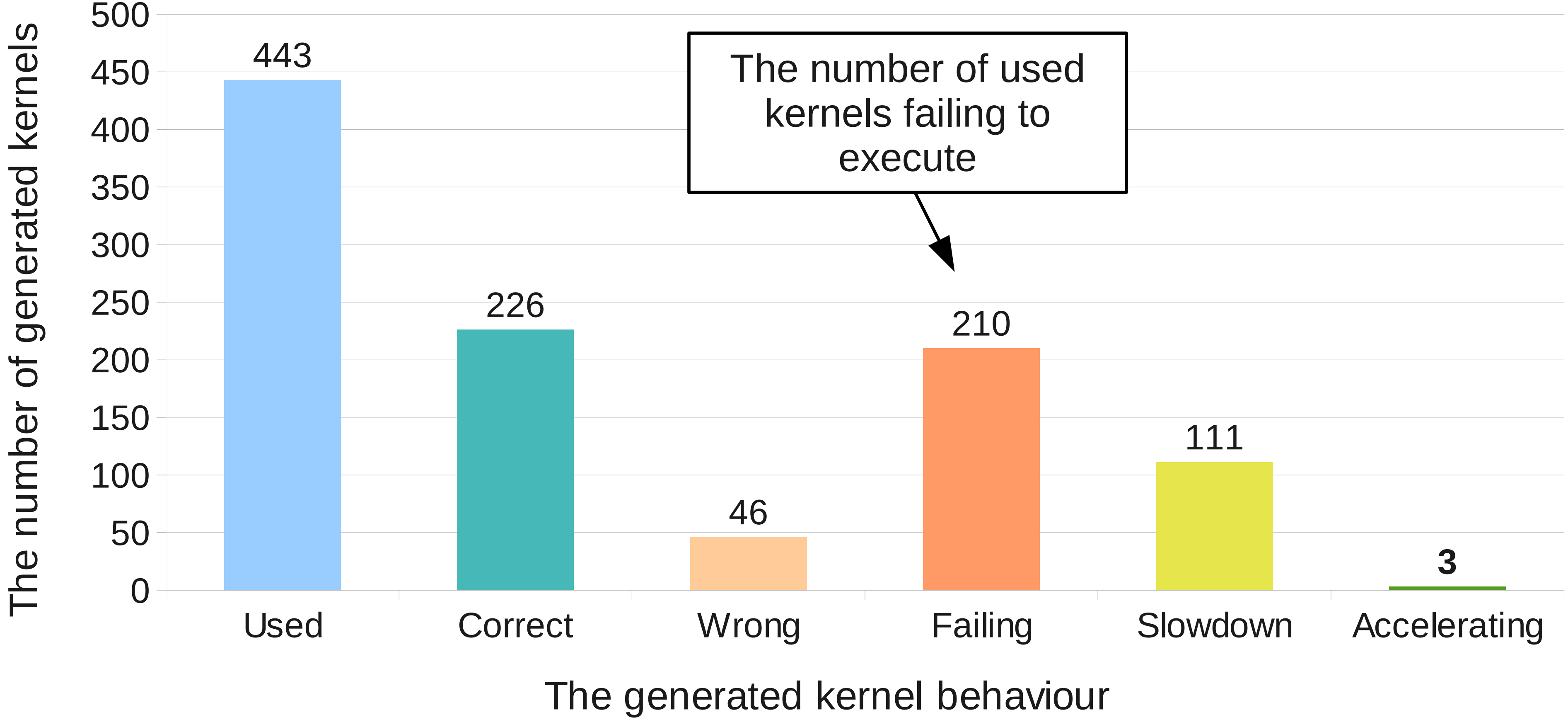


COSMO – coverage

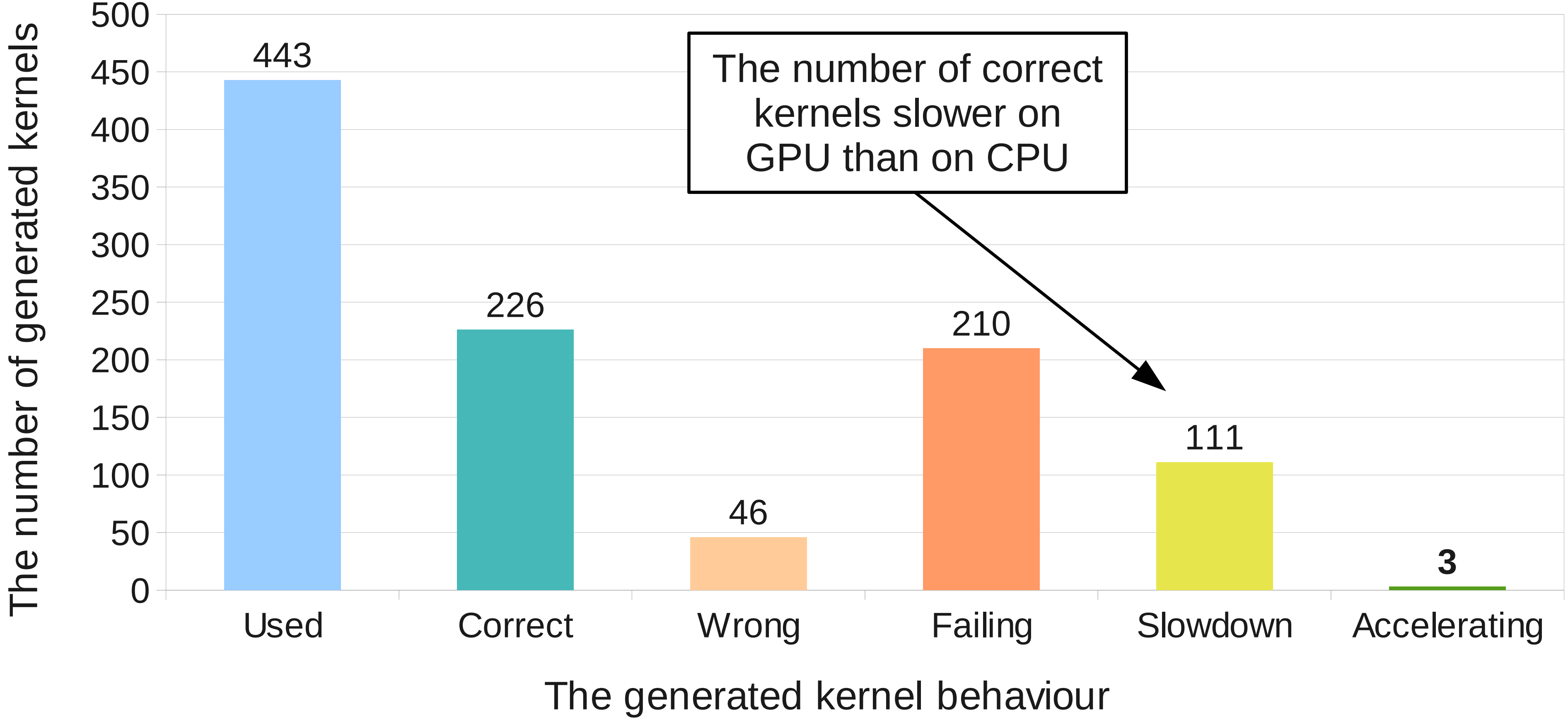


2011

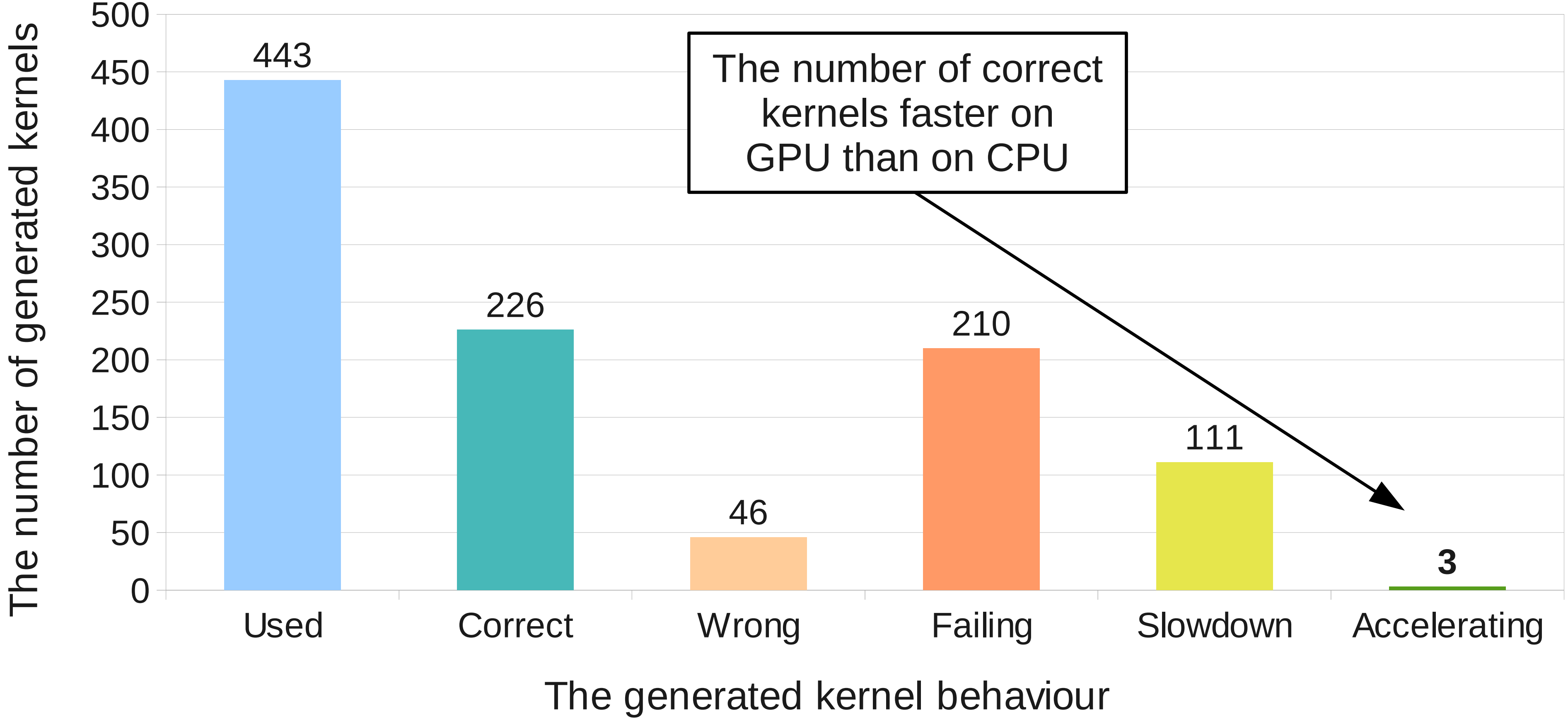
COSMO – coverage



COSMO – coverage

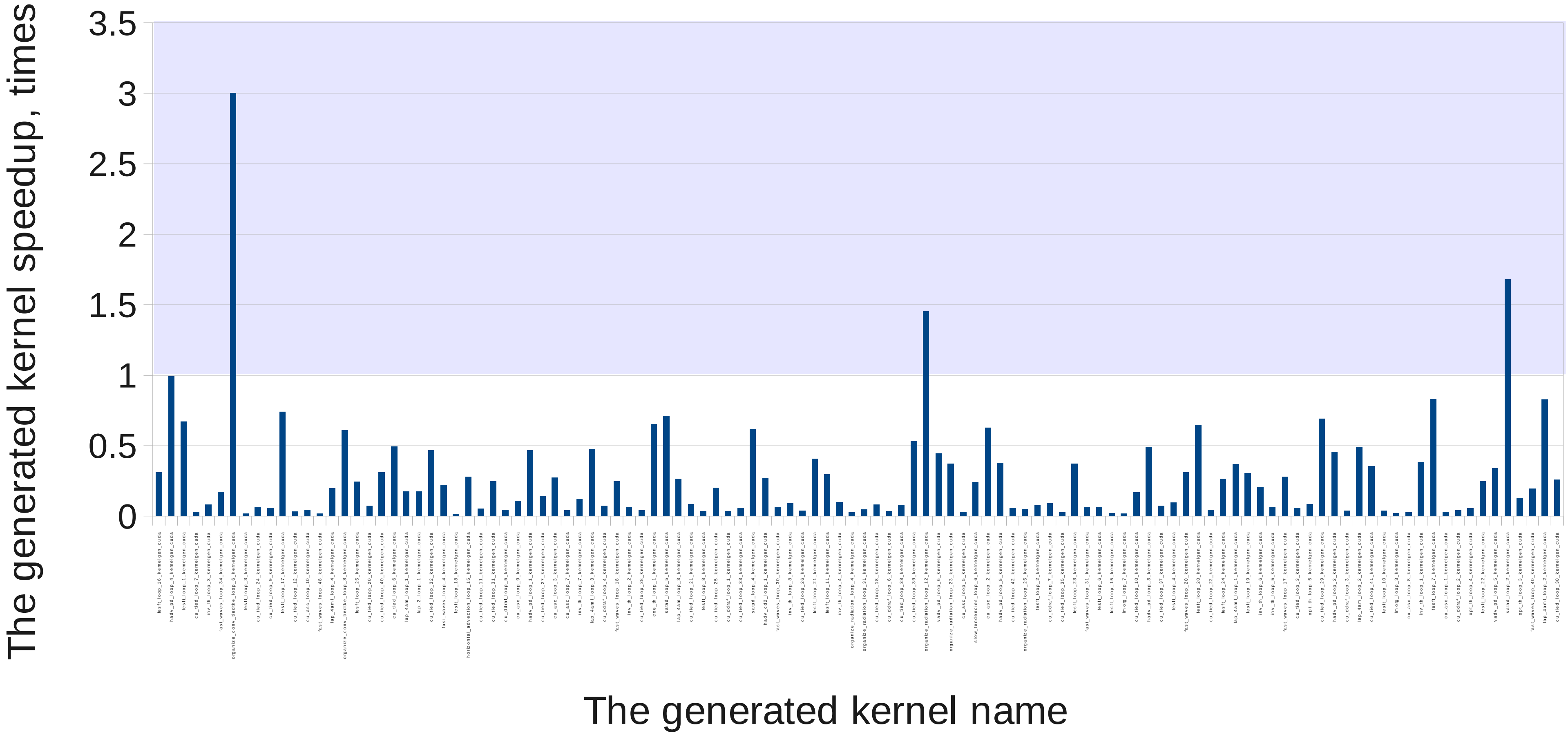


COSMO – coverage

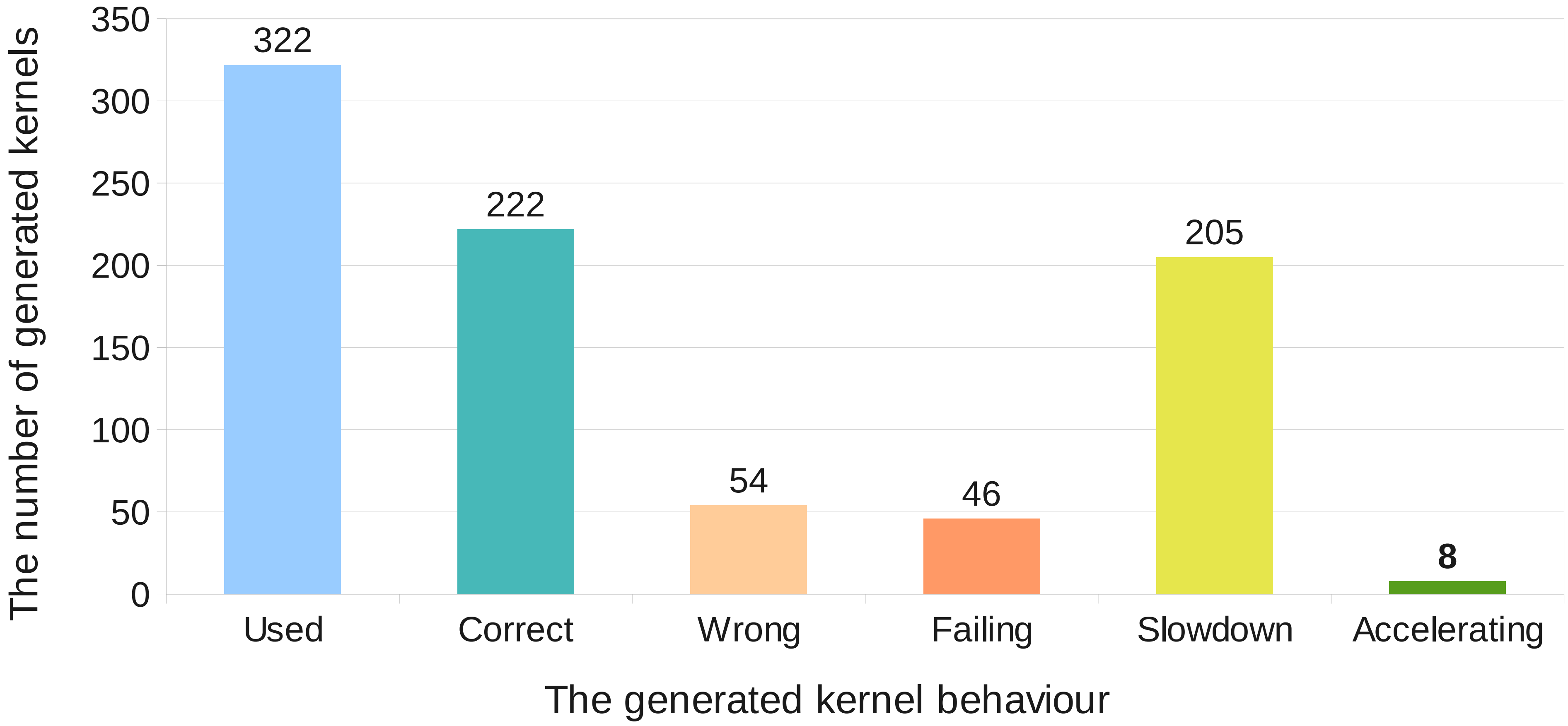


2011

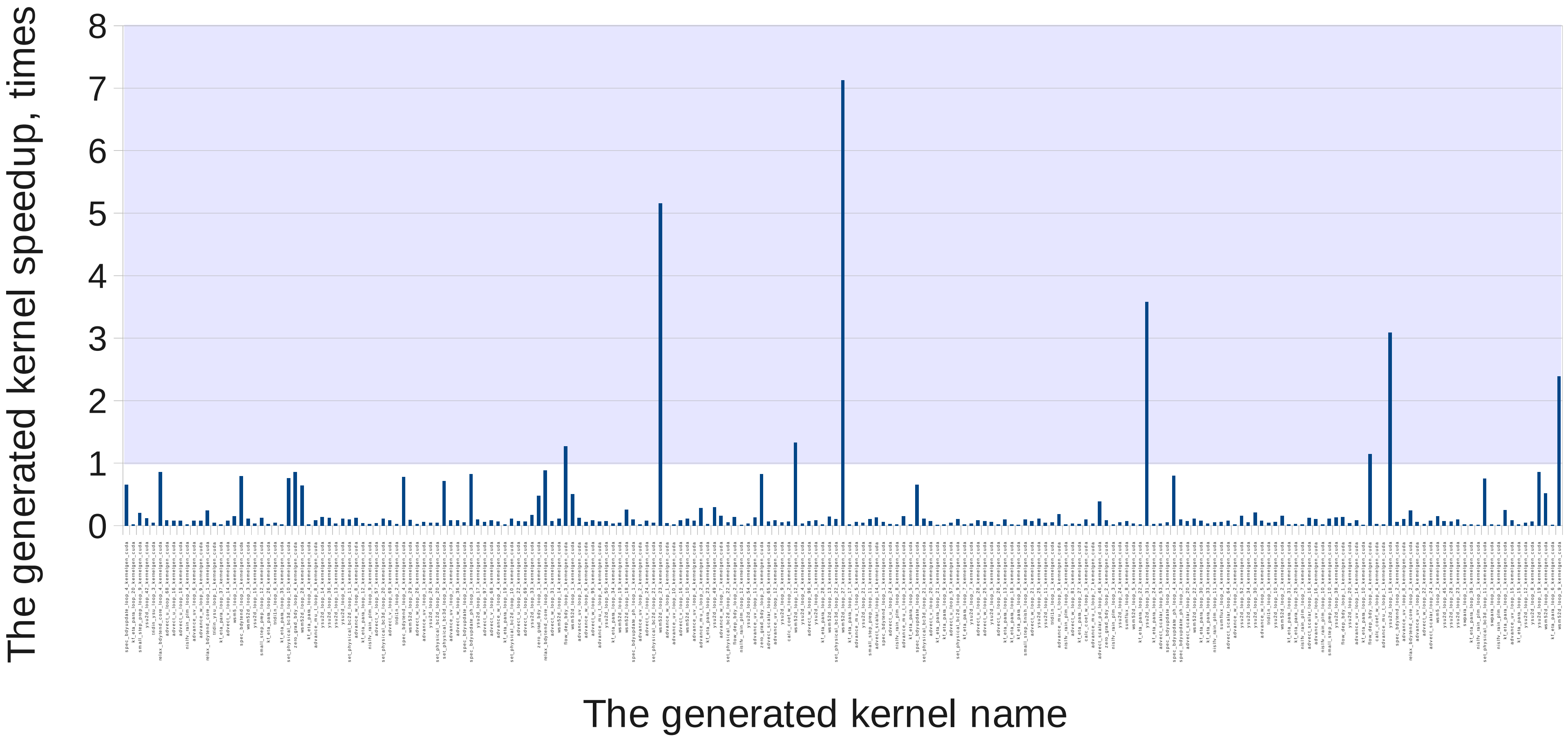
COSMO – performance



WRF – coverage



WRF – performance



2011

Why slowdown?

- **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

6. 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
- 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 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>

5. 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

2011

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

Collaboration

<u>We provide:</u>

- Source code
- User support, updates and bug fixes

<u>We need:</u>

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