



```
call gpu(  
  int ex, ...  
  n, float* out)  
  
  ! Compute absolute (i,j) index  
  ! of current GPU thread using  
  ! blockIdx.x * BLOCK_LENGTH  
  ! blockIdx.y * BLOCK_HEIGHT  
  ! Compute one data point  
  ! for the given  
  ! = 0.1f  
  ! 2.0f *
```

KernelGen naïve GPU kernels generation from Fortran source code

Dmitry Mikushin

Contents

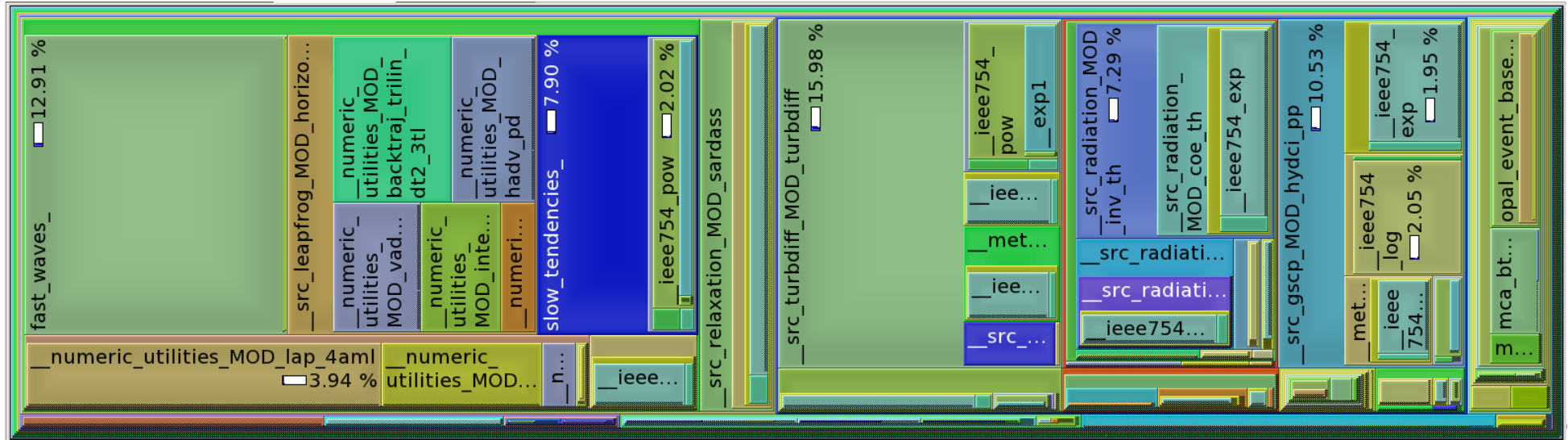
- Motivation, target, analysis
- Assembling our own toolchain
- Toolchain usecase: axpy example
- Development schedule

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 ($\sim 10\%$), 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 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

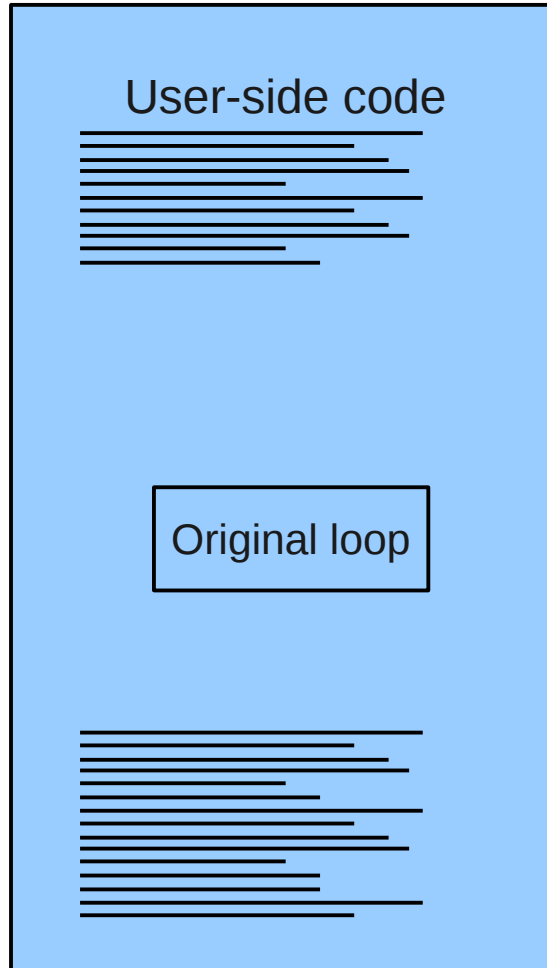
Ingredients

- **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

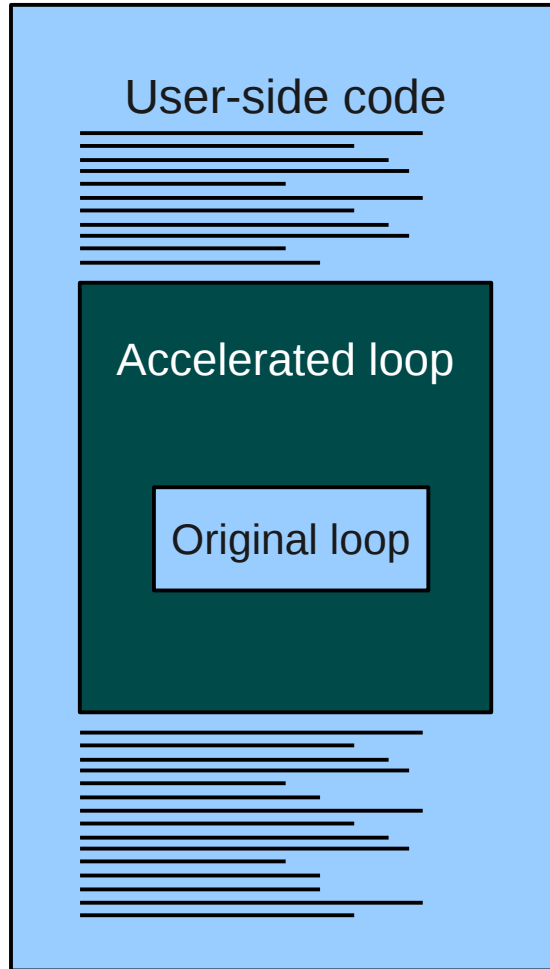
- 1) Make up the rough version of the full toolchain first, focus on improvements later
- 2) Use empirical tests where analysis is not yet sufficient (e.g. for identifying parallel loops)
- 3) Focus on best compiled kernels yield (code coverage) for COSMO and other models
- 4) Implement optimizations later

Runtime workflow



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

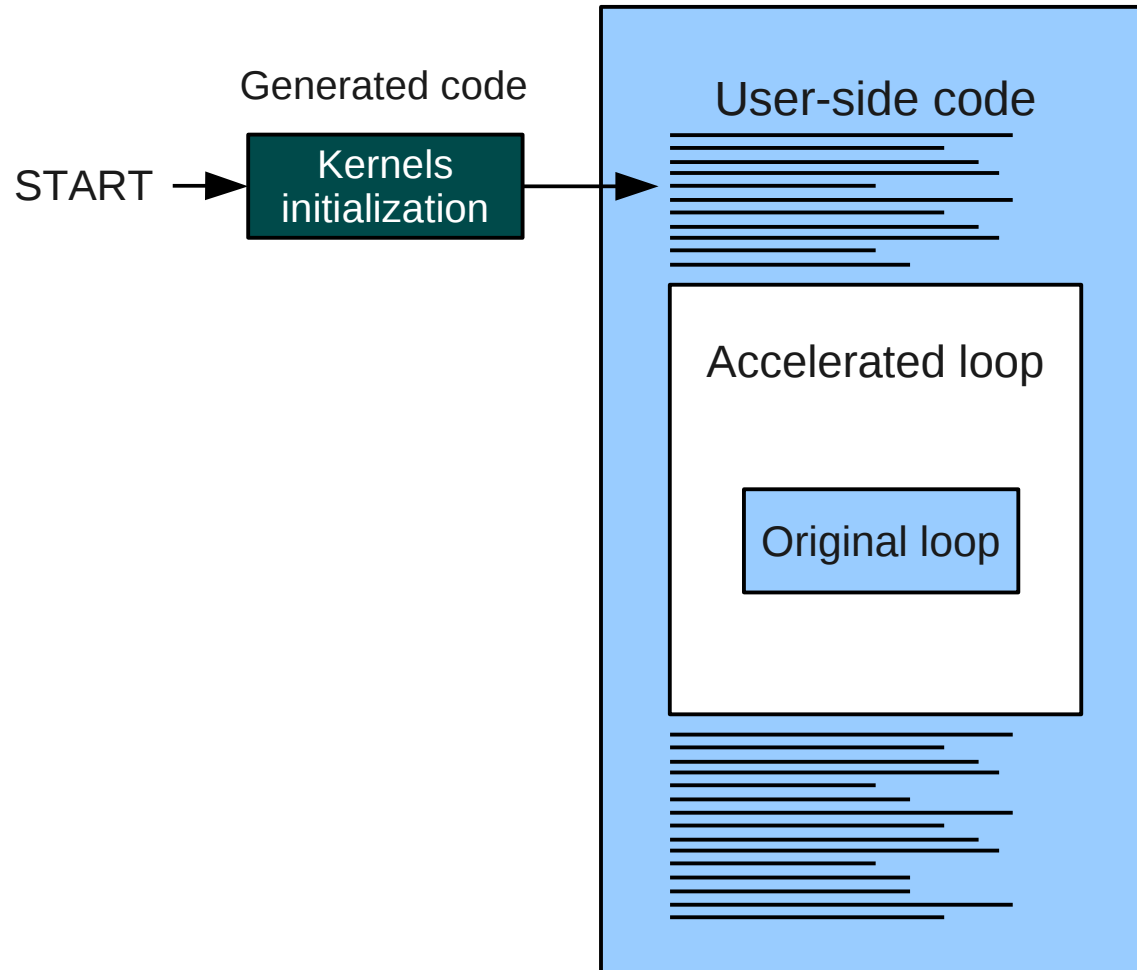
Runtime workflow



Equivalent device code is generated for suitable loops.

(see “Code generation workflow” for details)

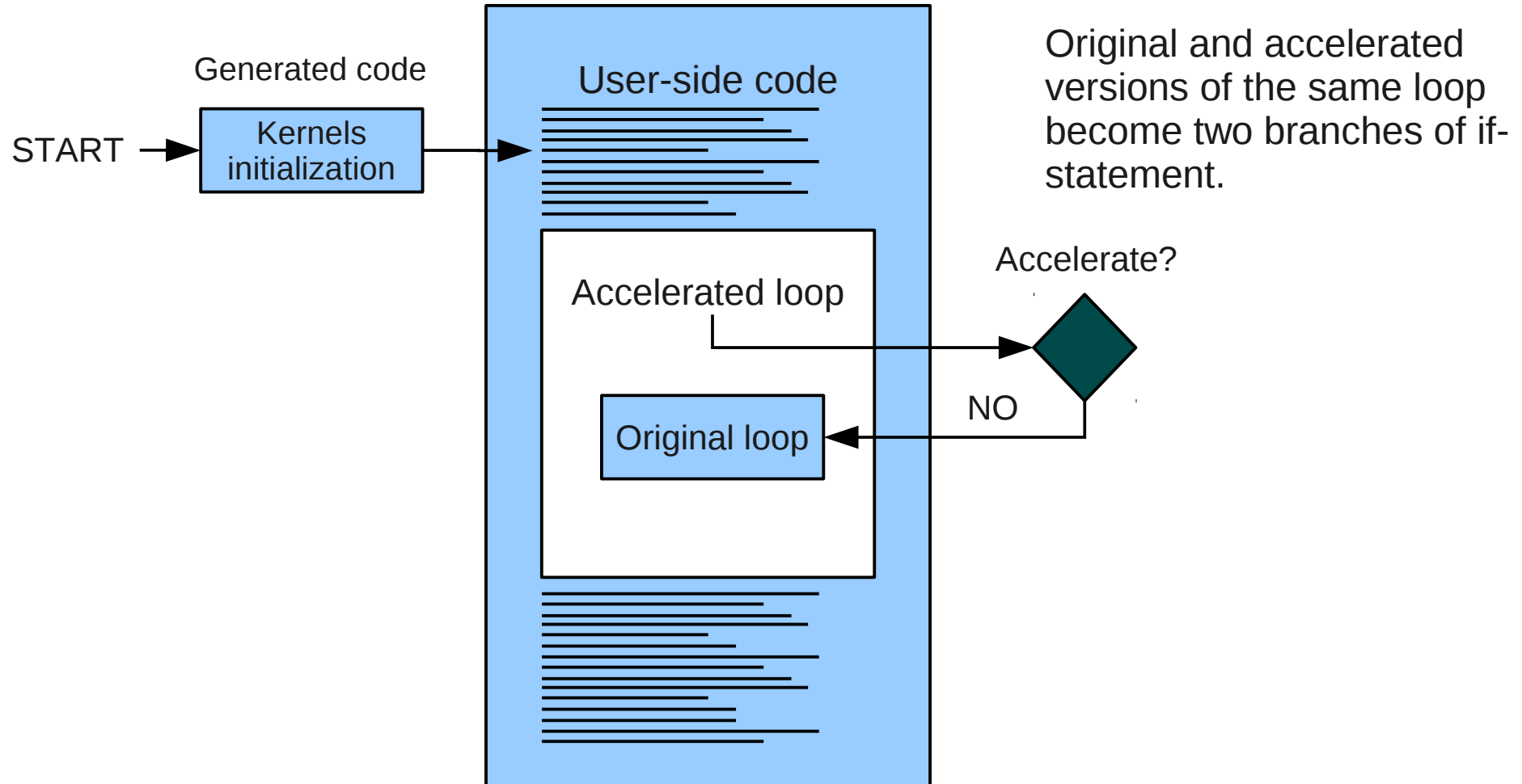
Runtime workflow



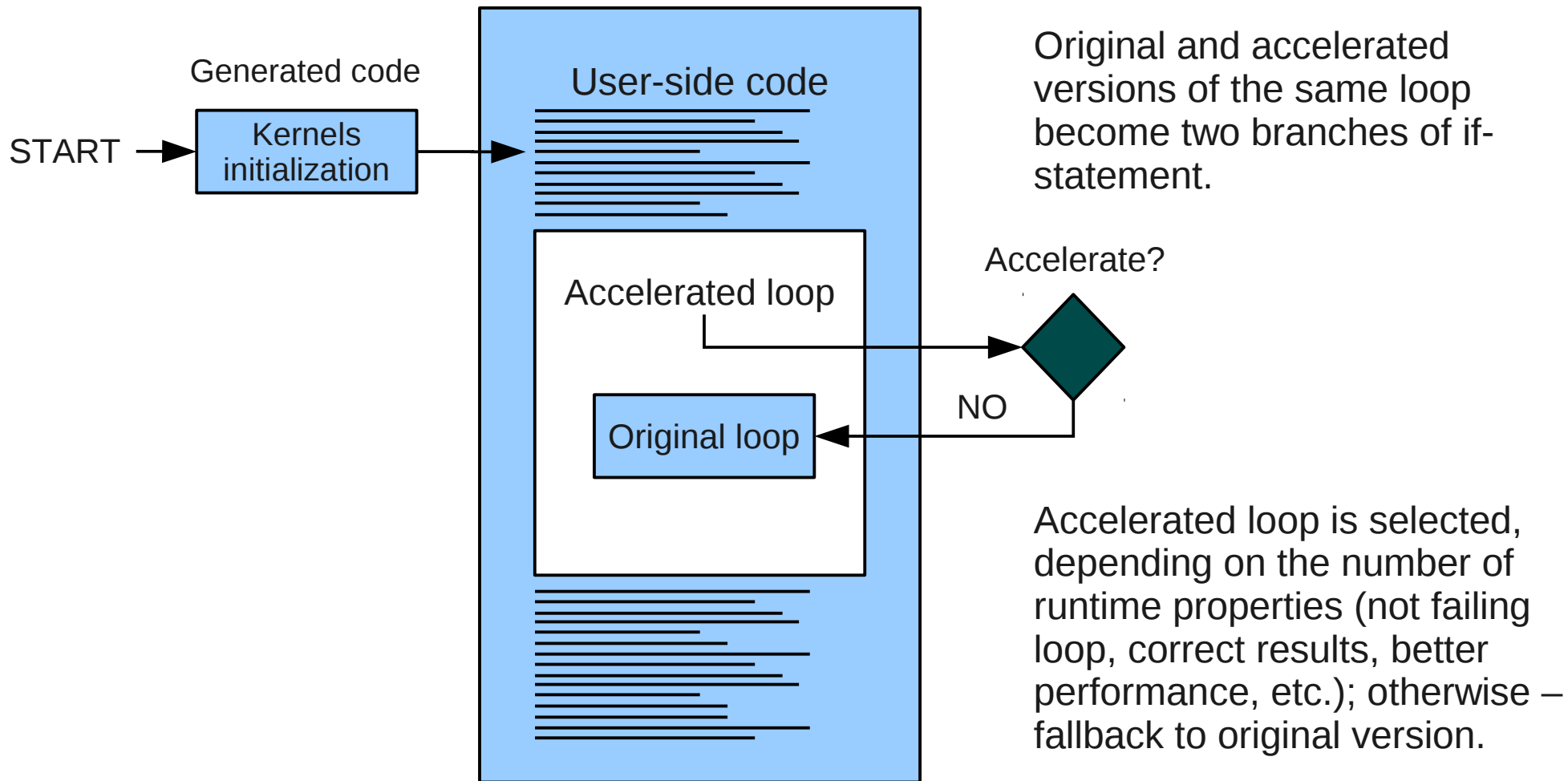
Equivalent device code is generated for suitable loops.

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

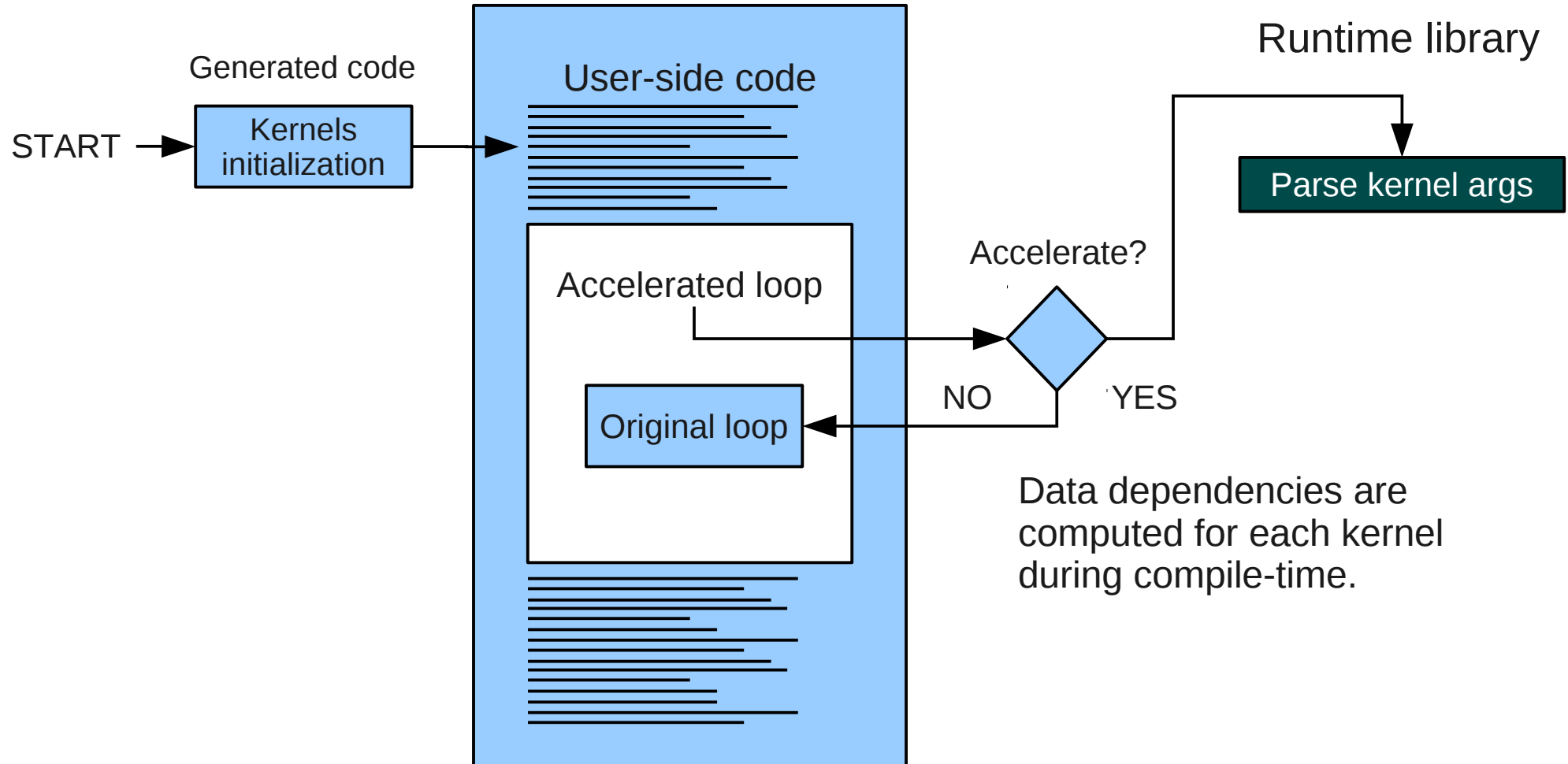
Runtime workflow



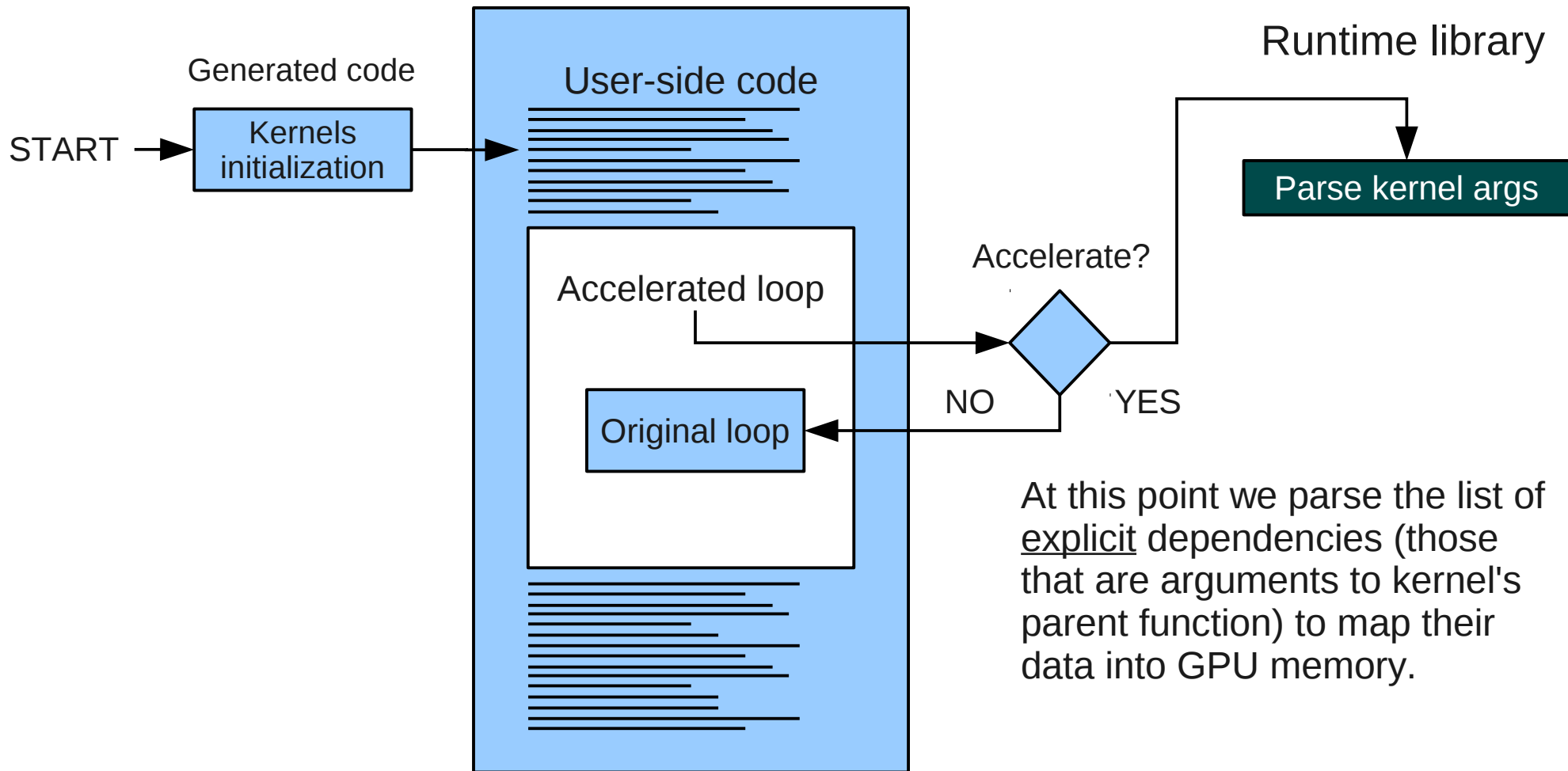
Runtime workflow



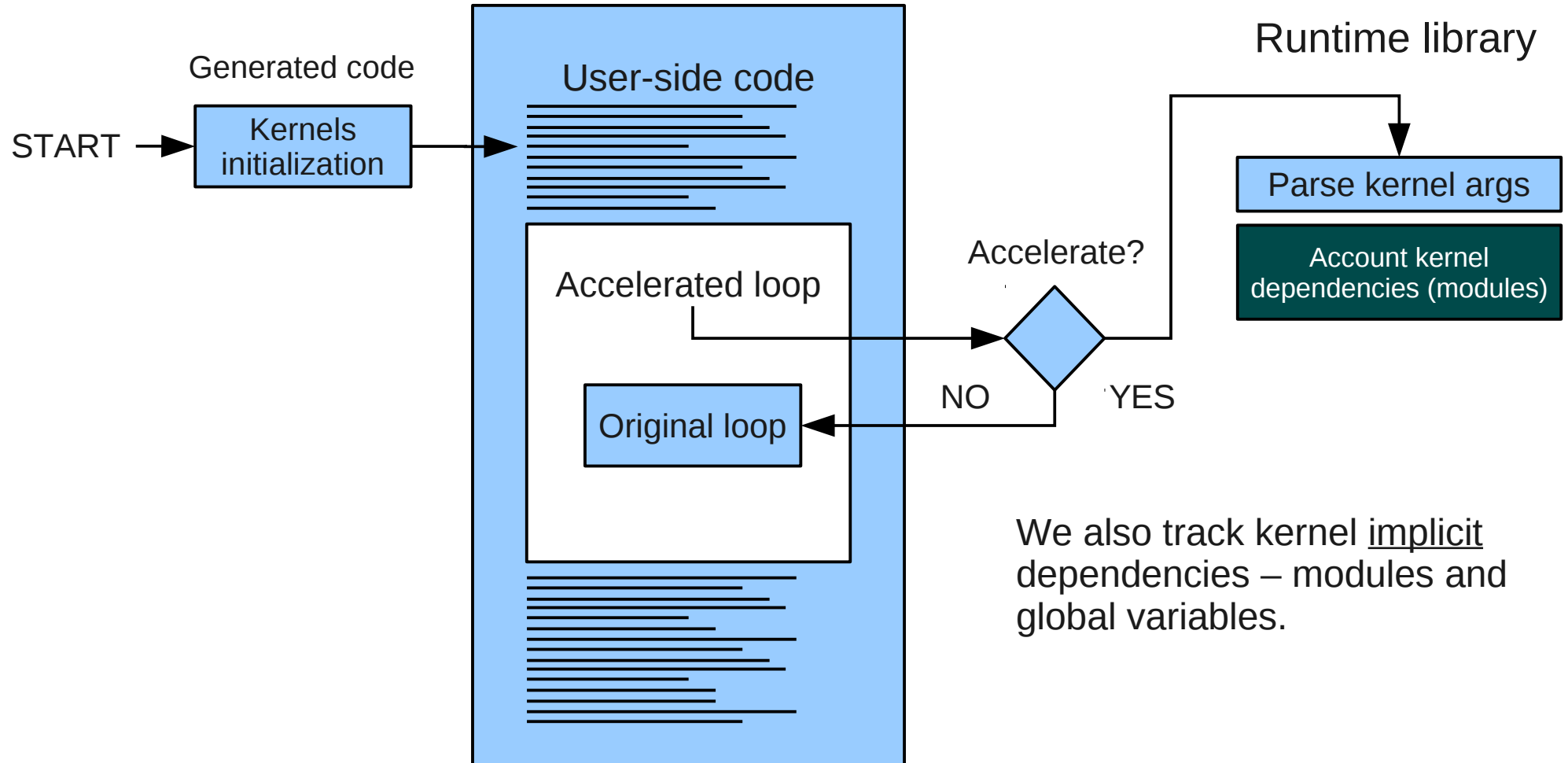
Runtime workflow



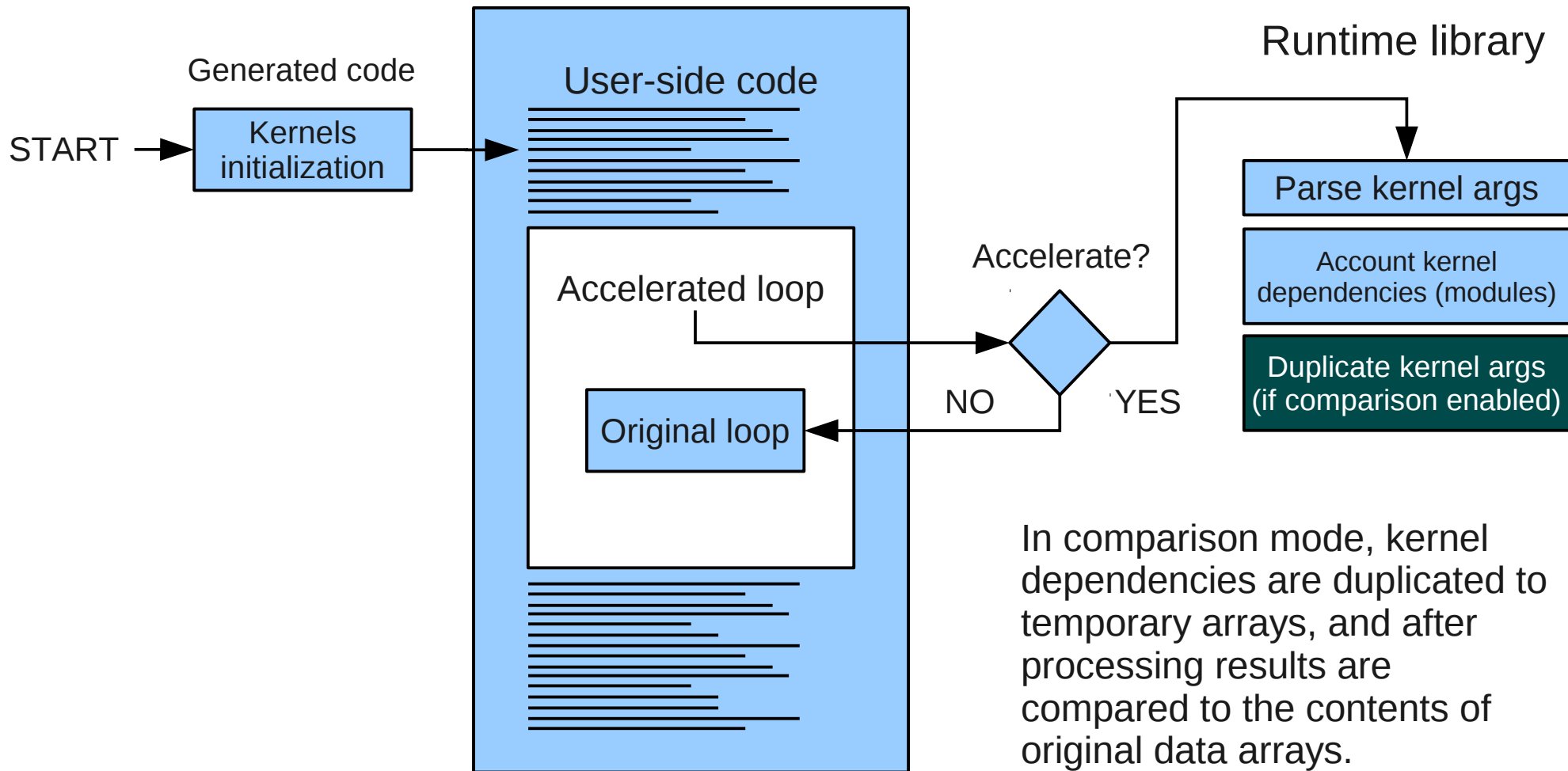
Runtime workflow



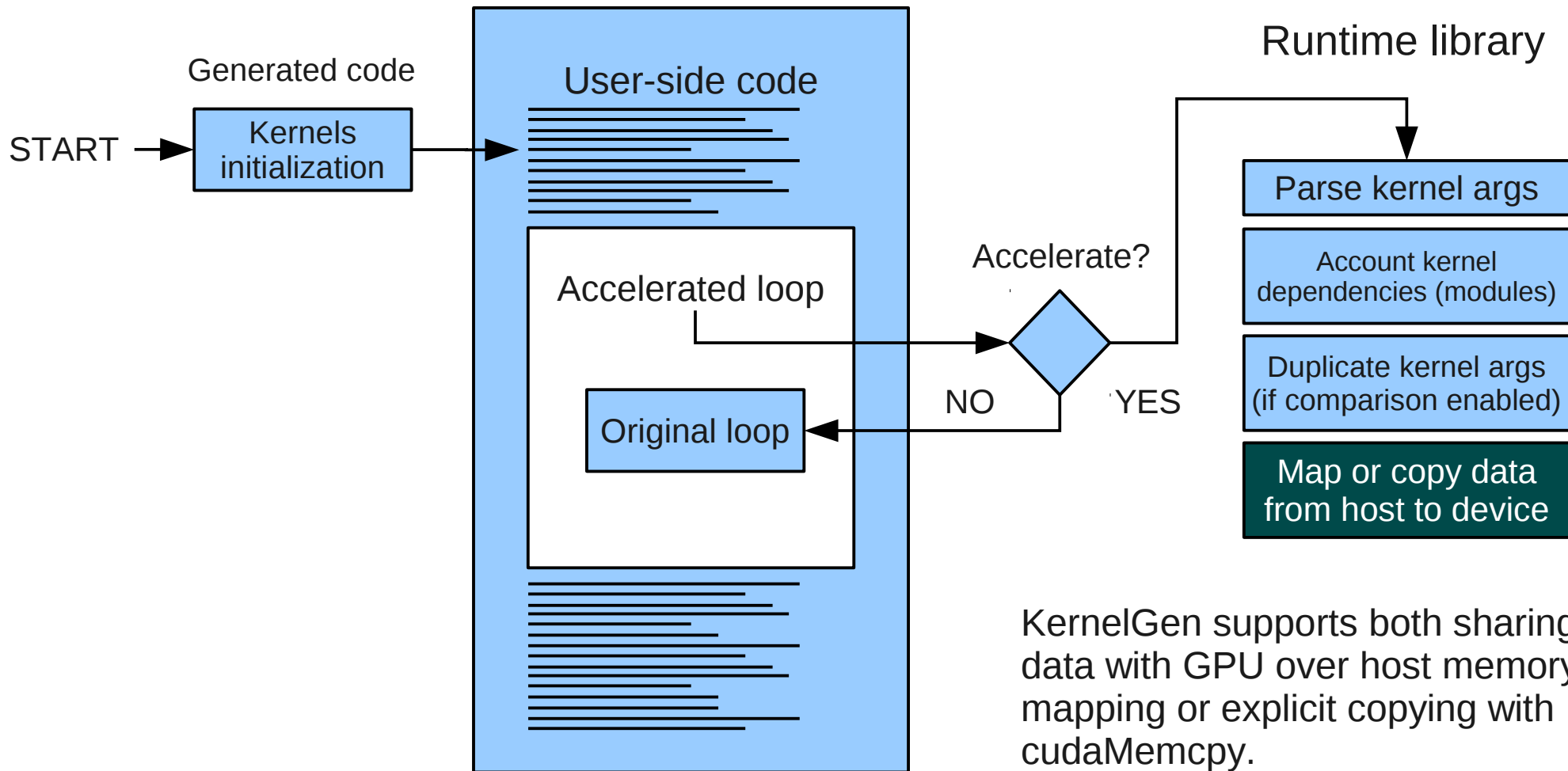
Runtime workflow



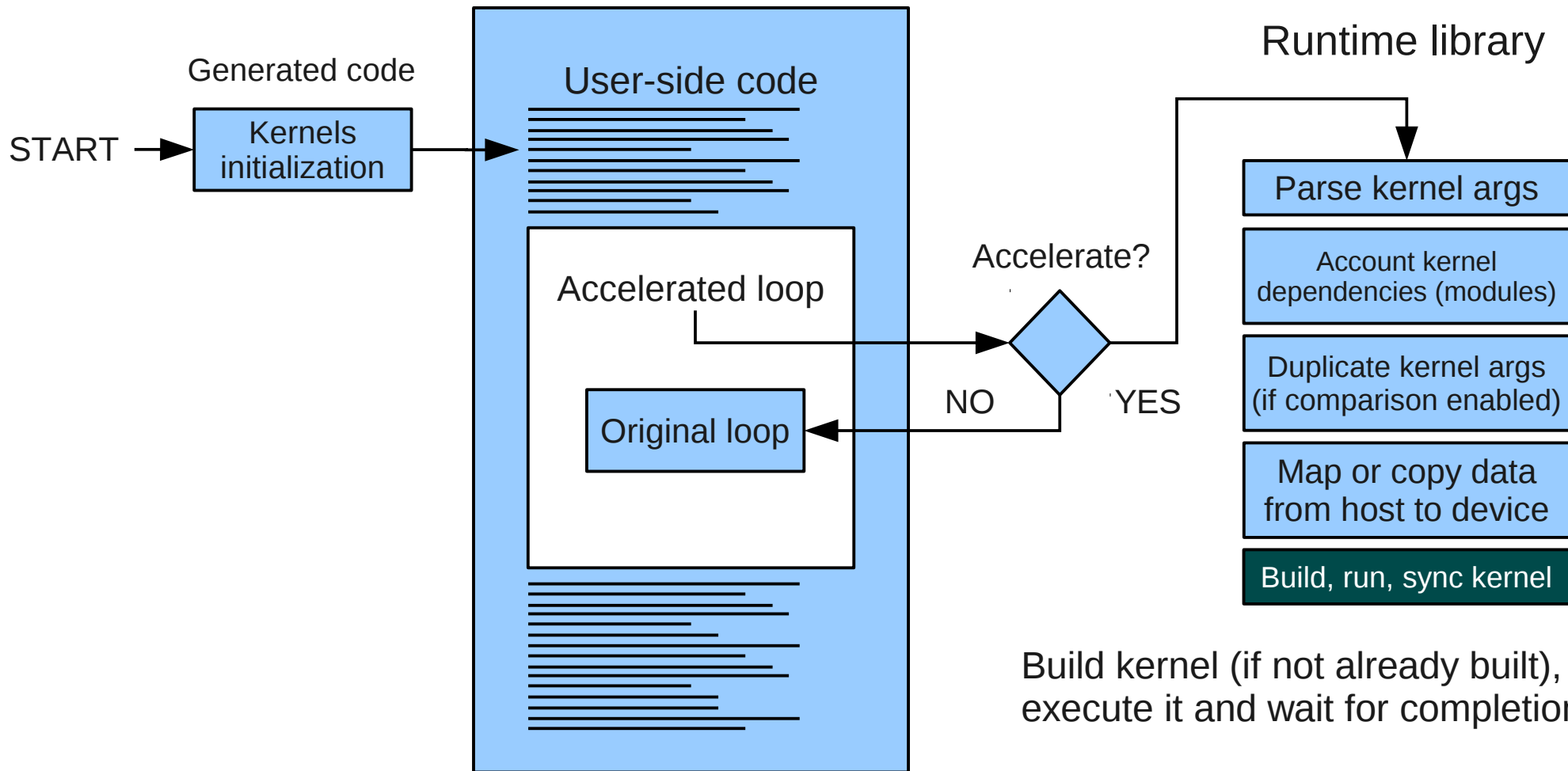
Runtime workflow



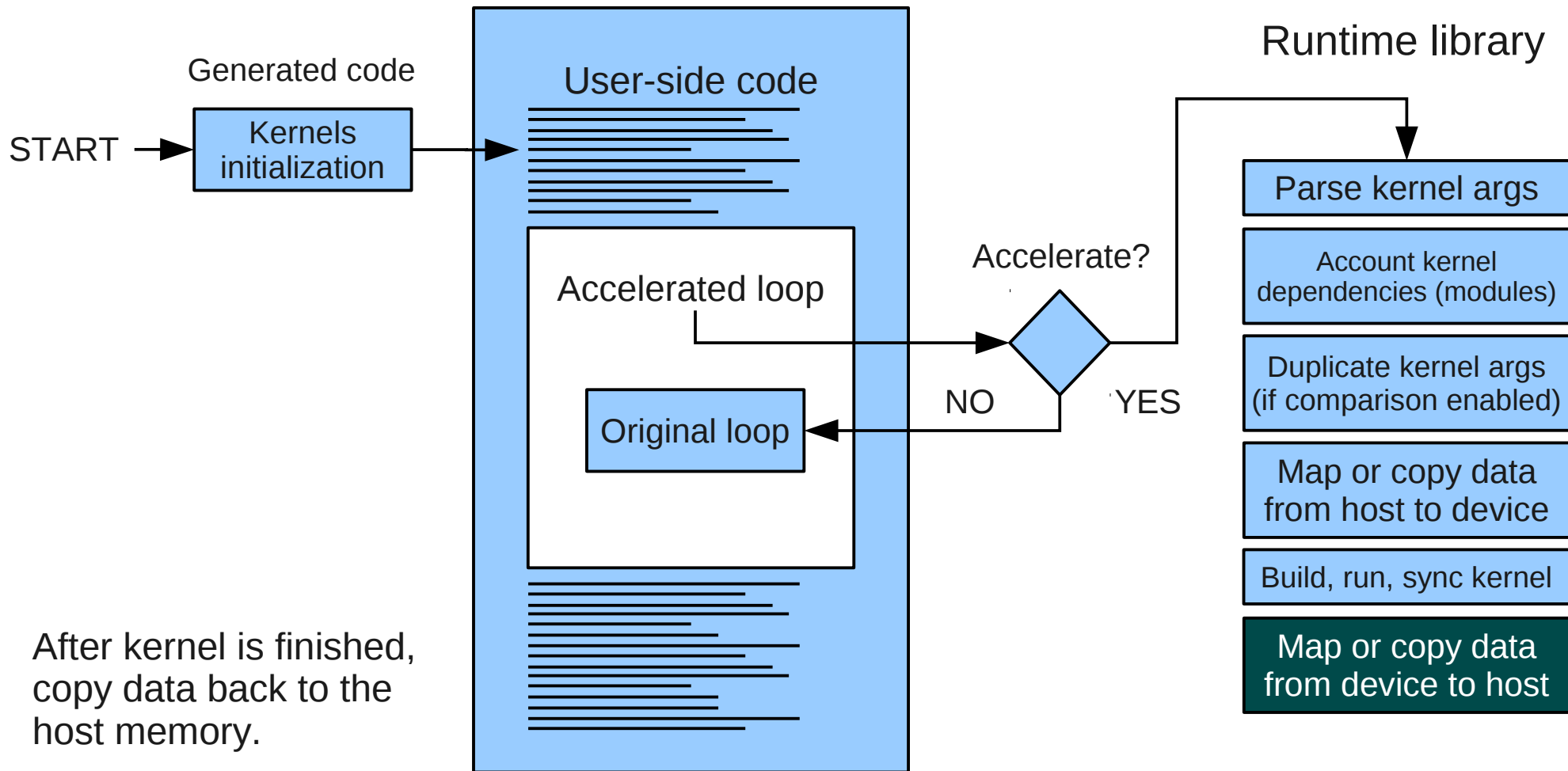
Runtime workflow



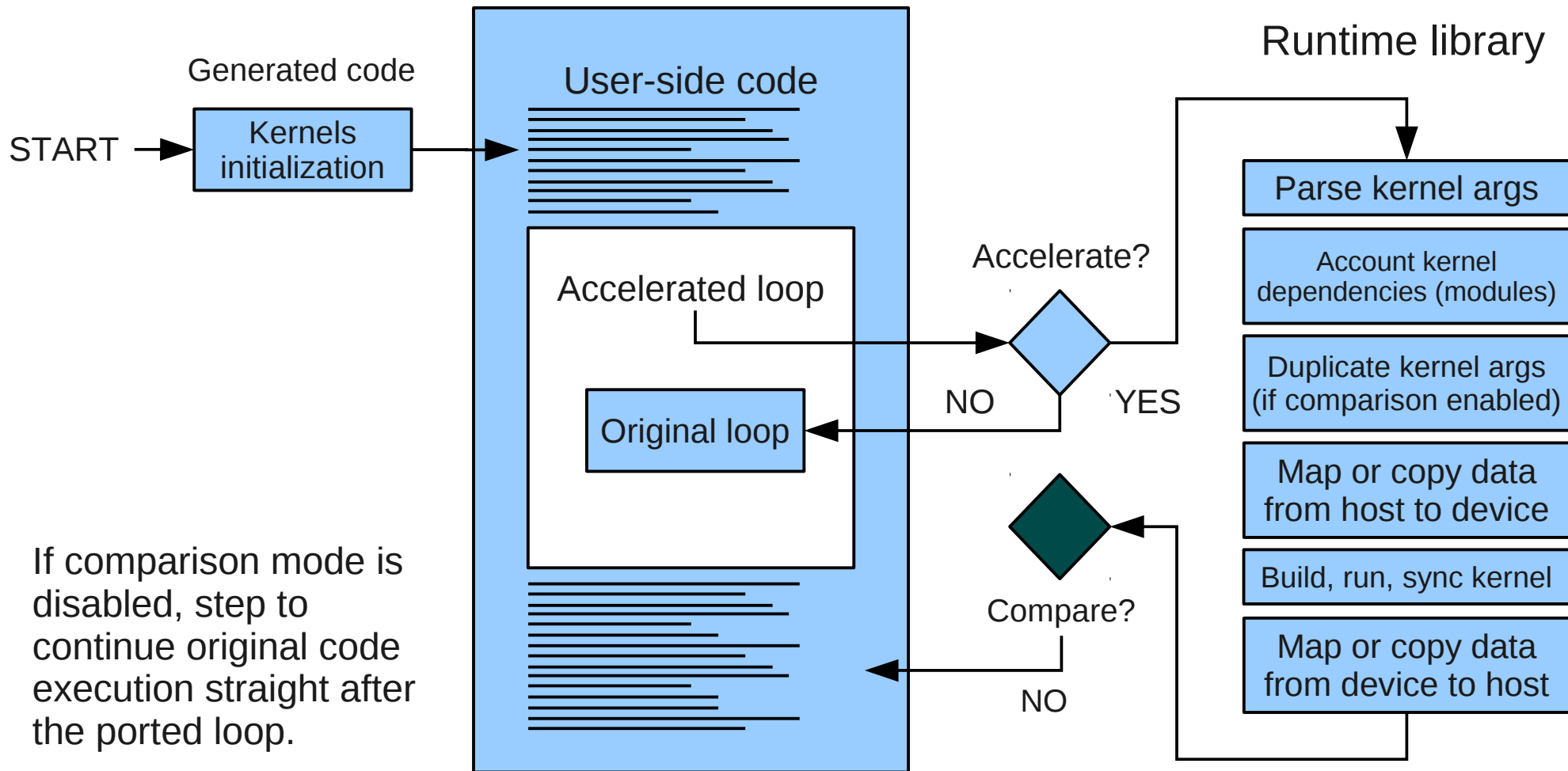
Runtime workflow



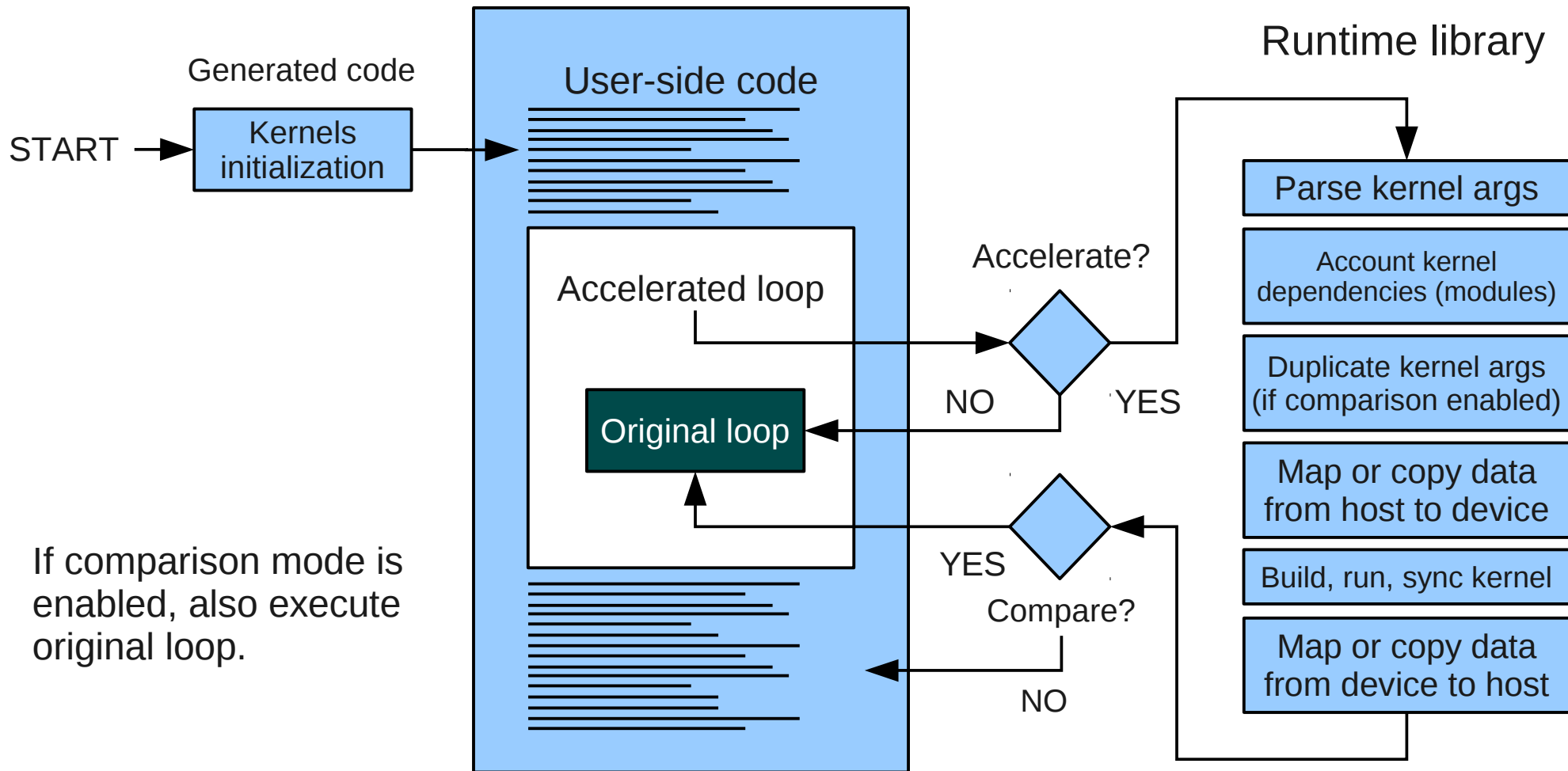
Runtime workflow



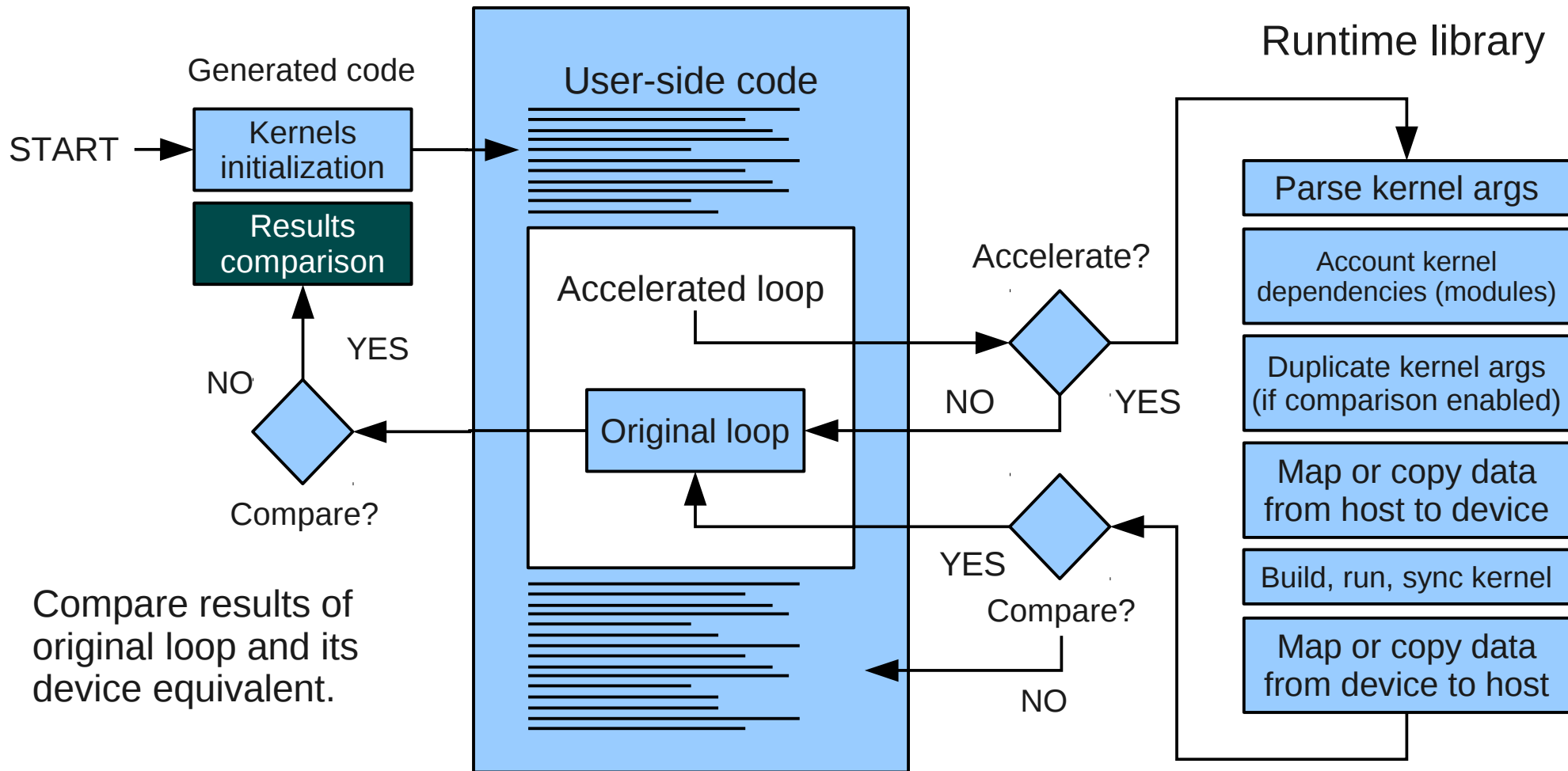
Runtime workflow



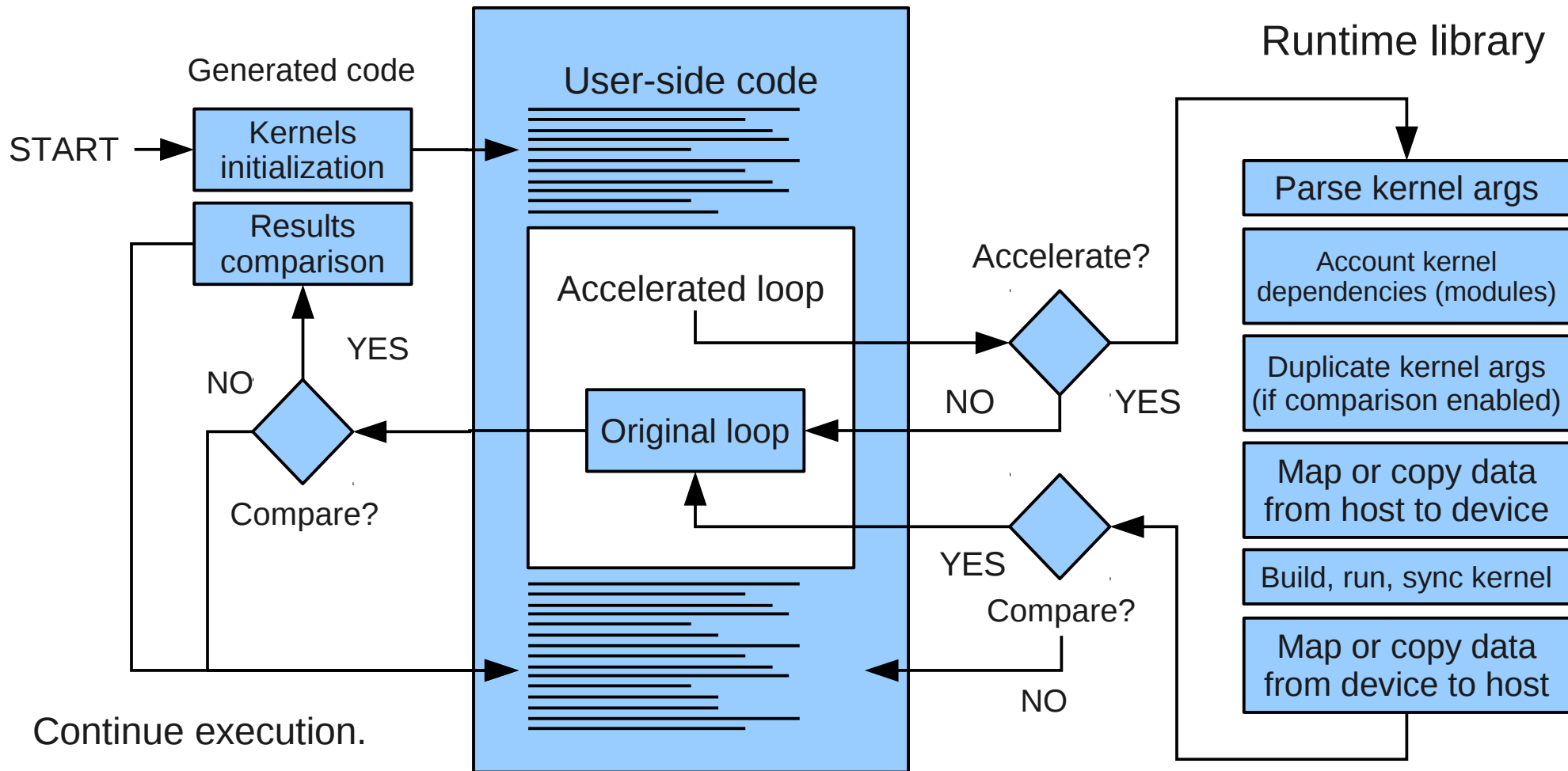
Runtime workflow



Runtime workflow



Runtime workflow

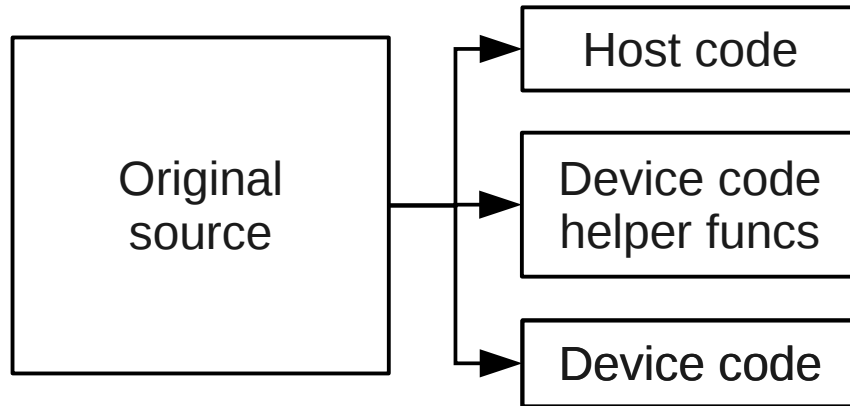


Code generation workflow

Two parts of code generation process:

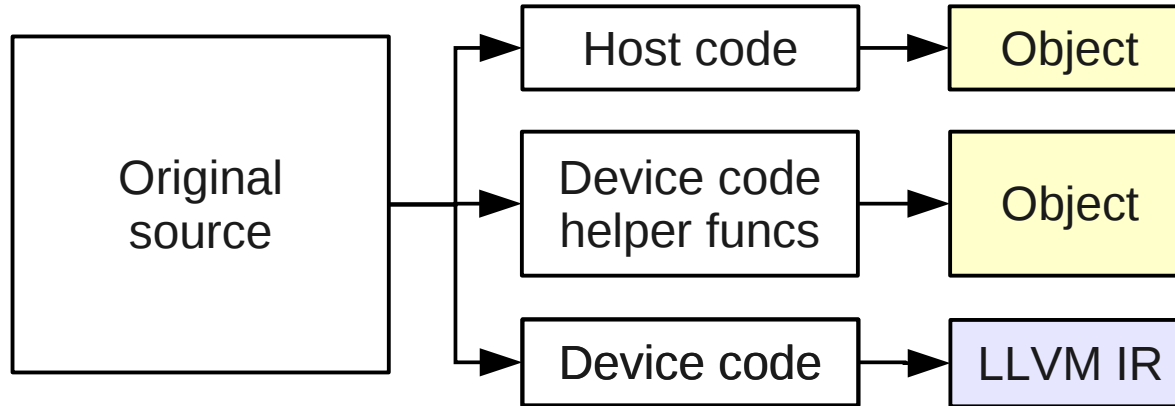
- **Compile time** – generate kernels strictly corresponding to original host loops
- **Runtime** – generate kernels, using additional info: inline external functions, optimize compute grid, etc.

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.

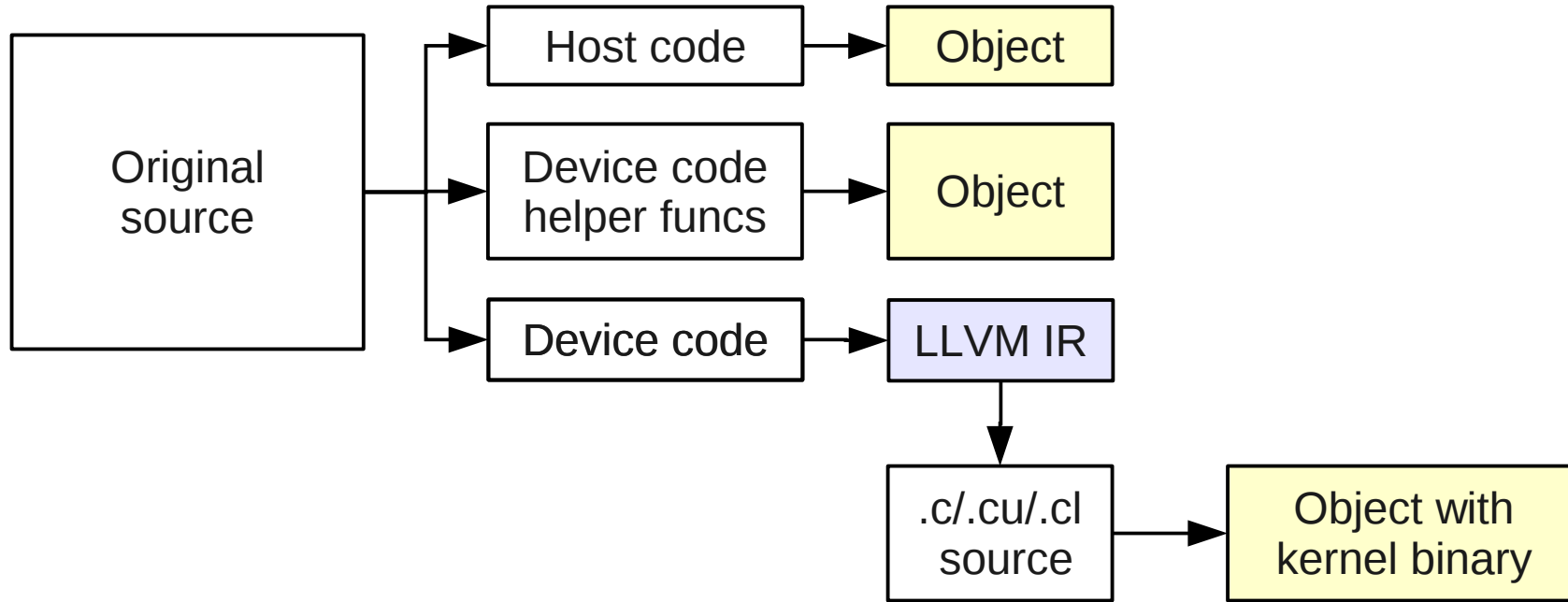
Code generation workflow (compile-time part)



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

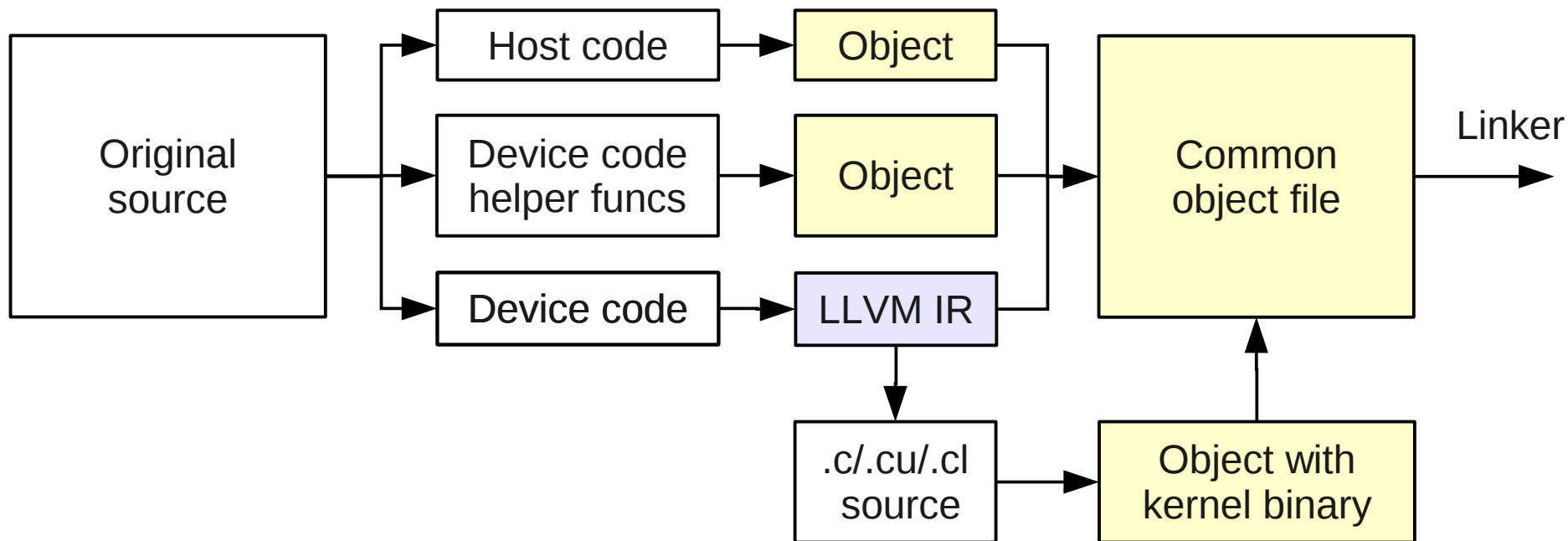
Device code is compiled into Low-Level Virtual Machine Intermediate representation (LLVM IR).

Code generation workflow (compile-time part)



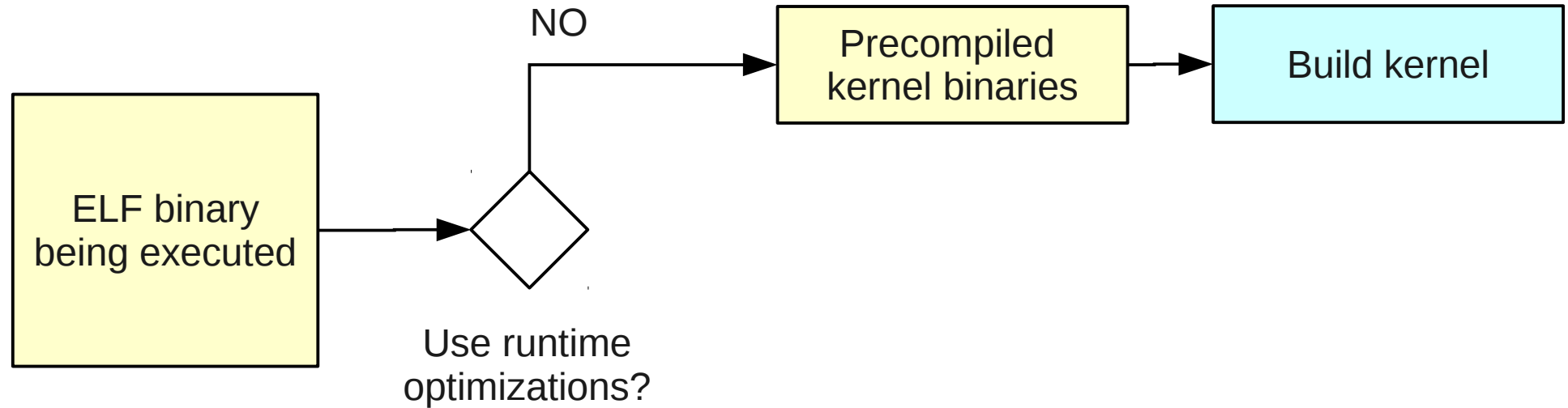
Code from LLVM IR is translated into C, CUDA or OpenCL using modified LLVM C Backend and compiled using the corresponding device compiler.

Code generation workflow (compile-time part)



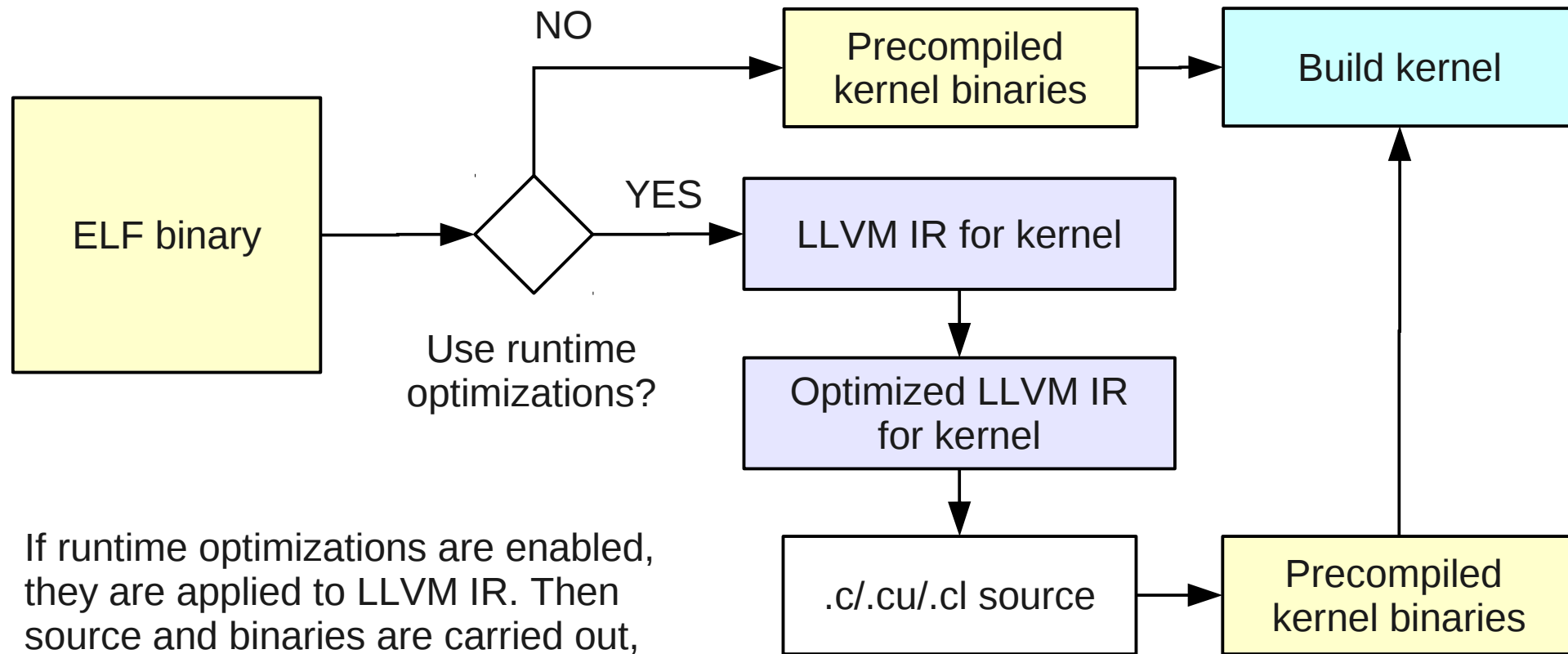
Finally, objects for all parts of the code are merged into single object to conserve “1 source → 1 object” layout. LLVM IR is also embedded into resulting object.

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. 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
```

Per-kernel config structure

```
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
```

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
```

```
call kernelgen_
```

```
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.

0,

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(
    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
!$KERNELGEN
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$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
```

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
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((iand(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (kernelgen_get_last_error() .ne. 0)) then
!$KERN
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
!$KERNELGEN END CALL sincos_loop_1_kernelgen
endif
if ((and(sincos_loop_1_kernelgen_config%runmode, kernelgen_runmode_host) .eq. 1)
.or. (k .eq. 0)) then
!$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
```

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
```

```
!$KERN
```

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

```
!$acc end region
```

```
end subroutine sincos
```

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

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

```
implicit none  
interface
```

Kernel subroutine name is a decorated name of original loop function

```
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
```


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

```

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

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-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), lu, 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\t\"GCC: (GNU) 4.5.4 20110527 (prerelease) LLVM: 131968 \t\t\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;
}
```

Steps 6-8

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

6

```
nvcc -g -c axpy.axy_loop_1_gforscale.F90.bc.cu -D__CUDA_DEVICE_FUNC__ -G -o  
axpy.axy_loop_1_gforscale.F90.o
```

7

```
gfortran -g -c axpy.host.F90 -D__CUDA_DEVICE_FUNC__ -ffree-line-length-none  
-l/opt/kgem/include -o axpy.host.F90.o
```

8

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

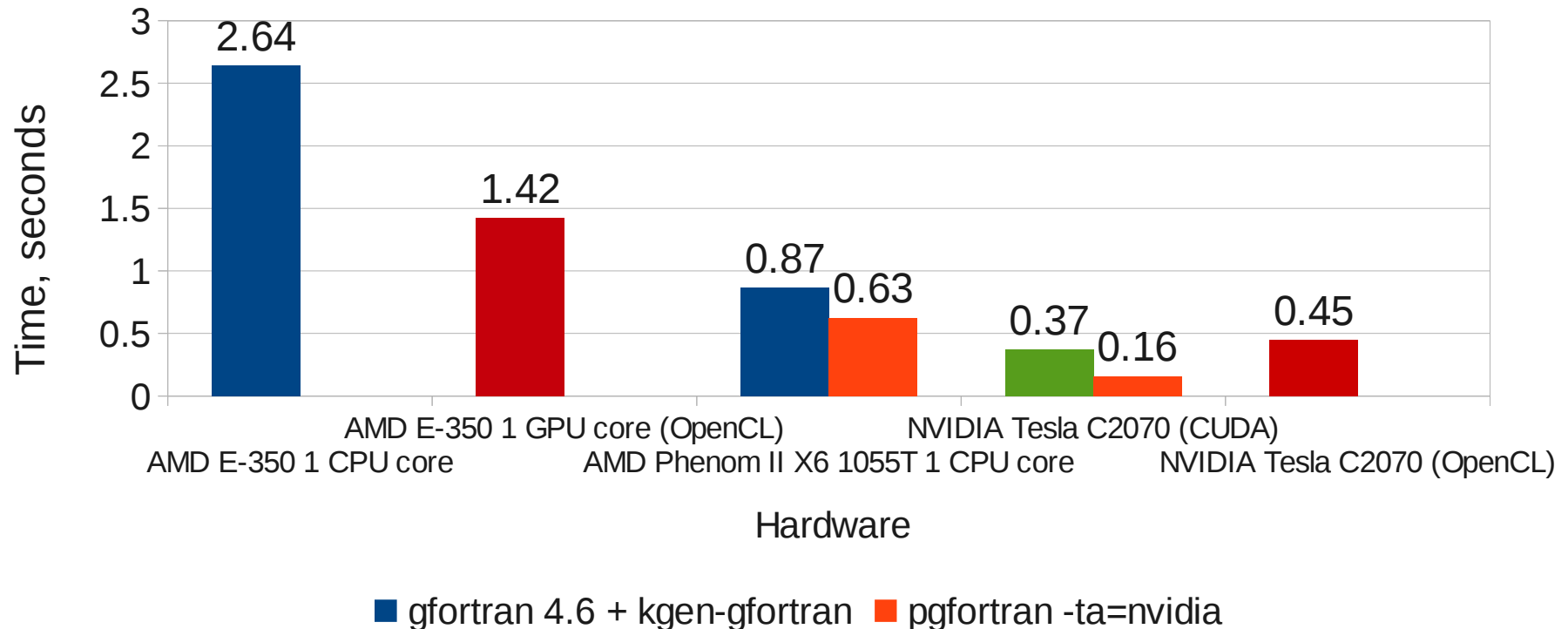
Use GPU kernel if the corresponding environment variable set to 1
(extra debug output showing how arguments are mapped into device memory)

```
[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. Testing unoptimized generator

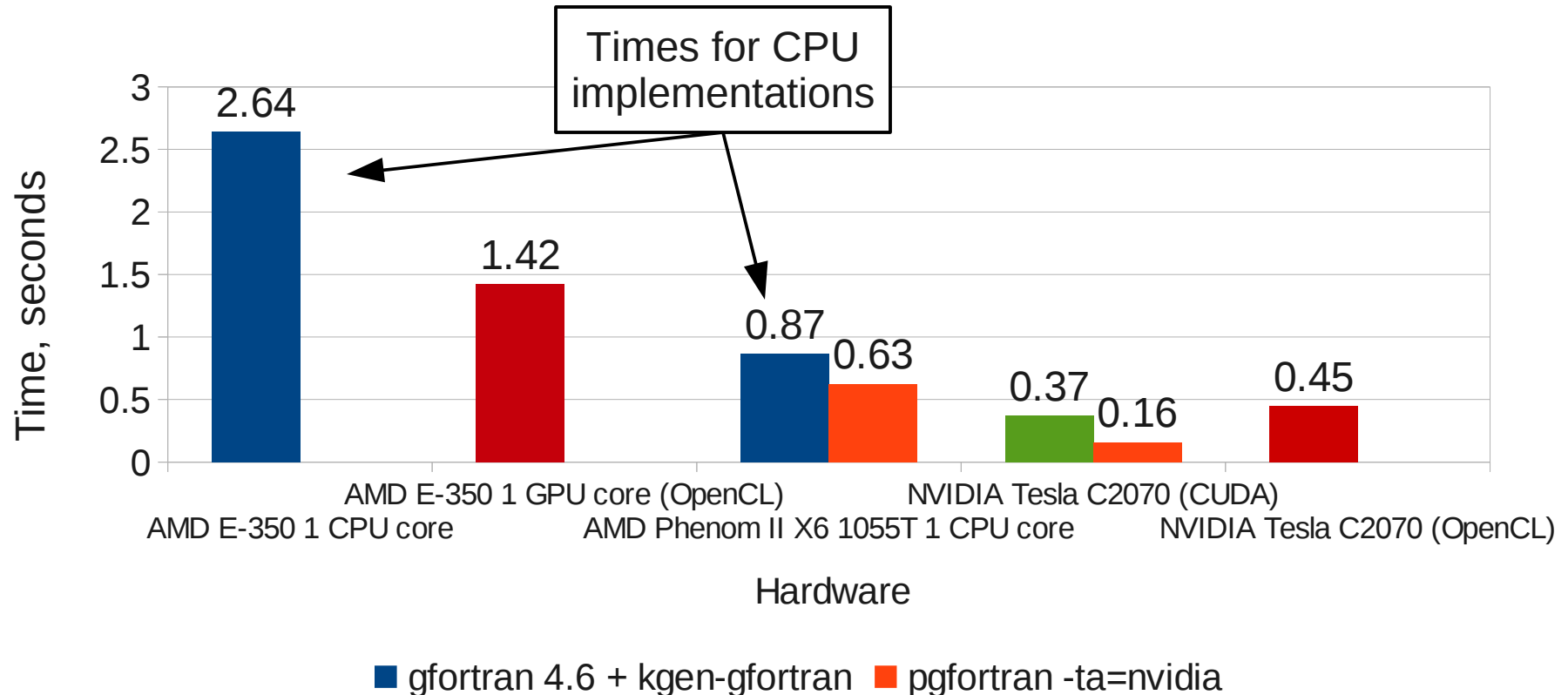
Example: sincos - performance

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



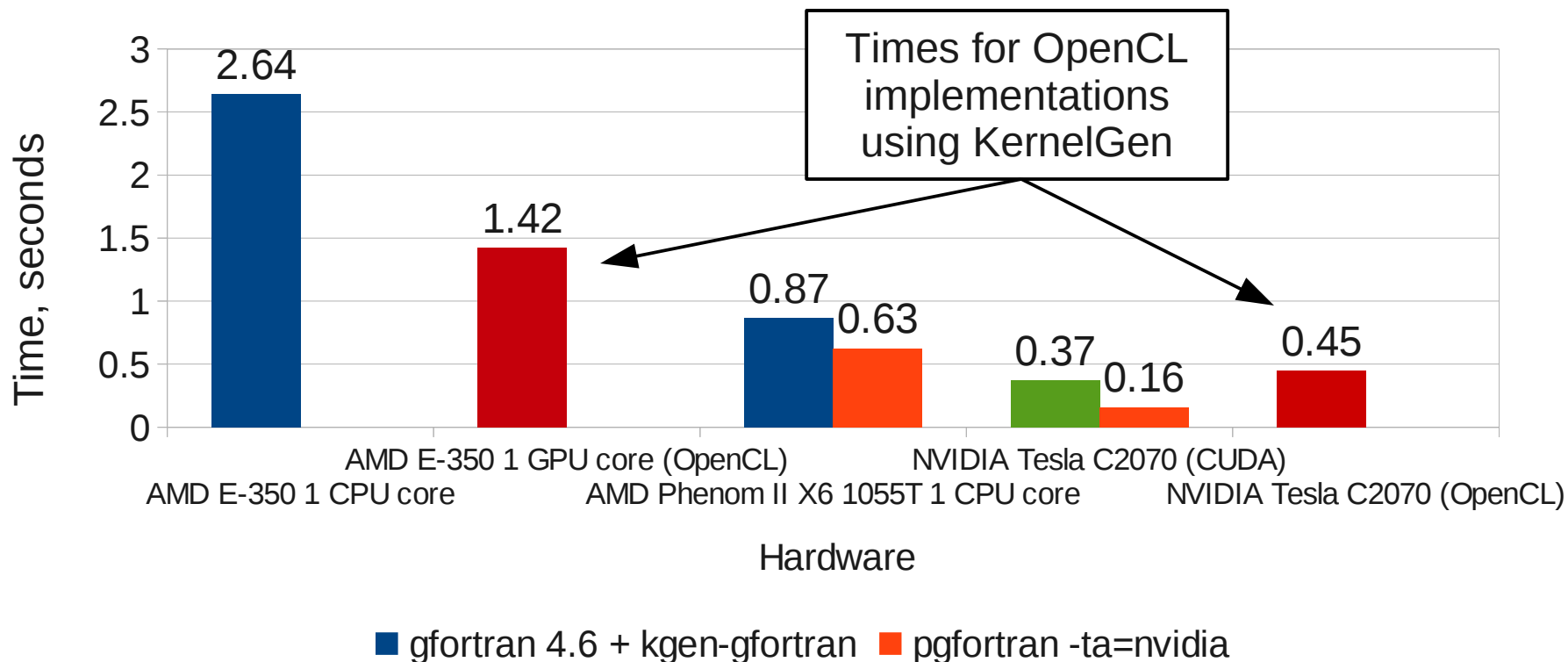
Example: sincos - performance

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



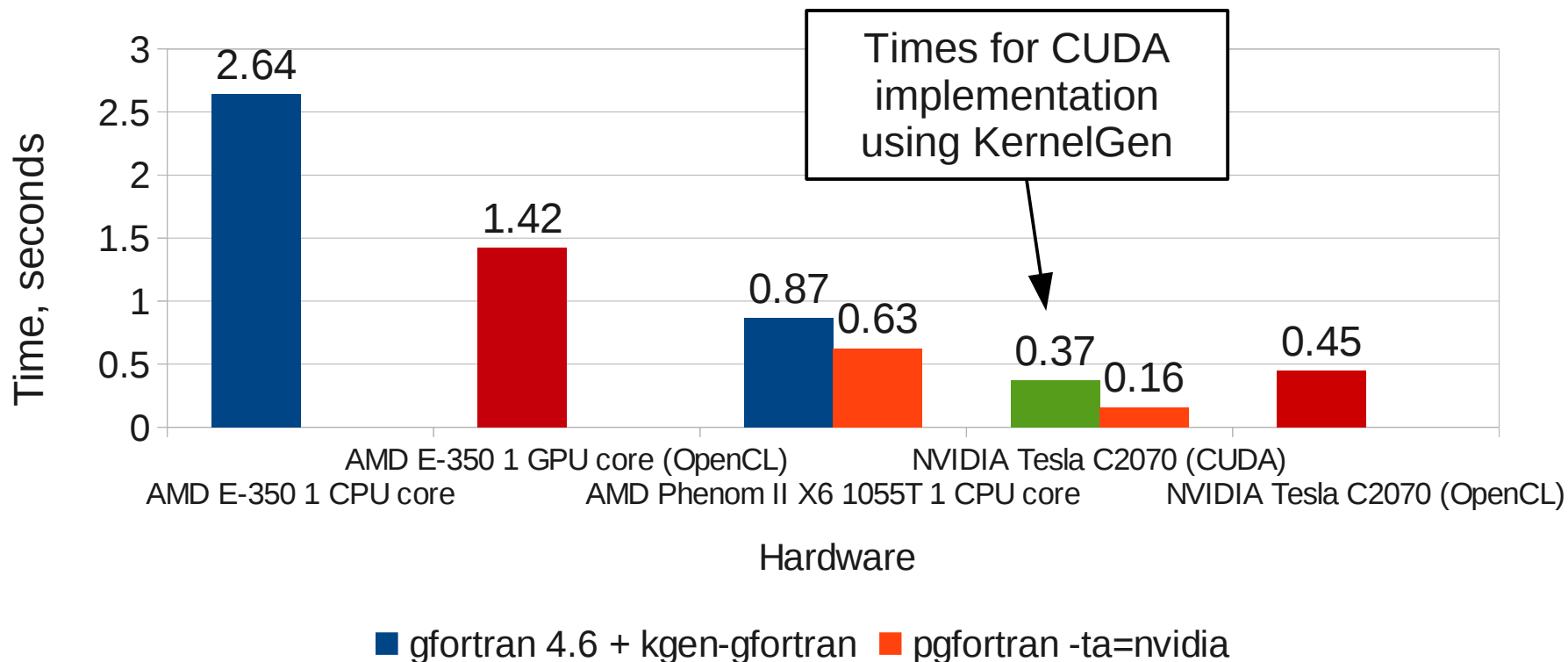
Example: sincos - performance

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



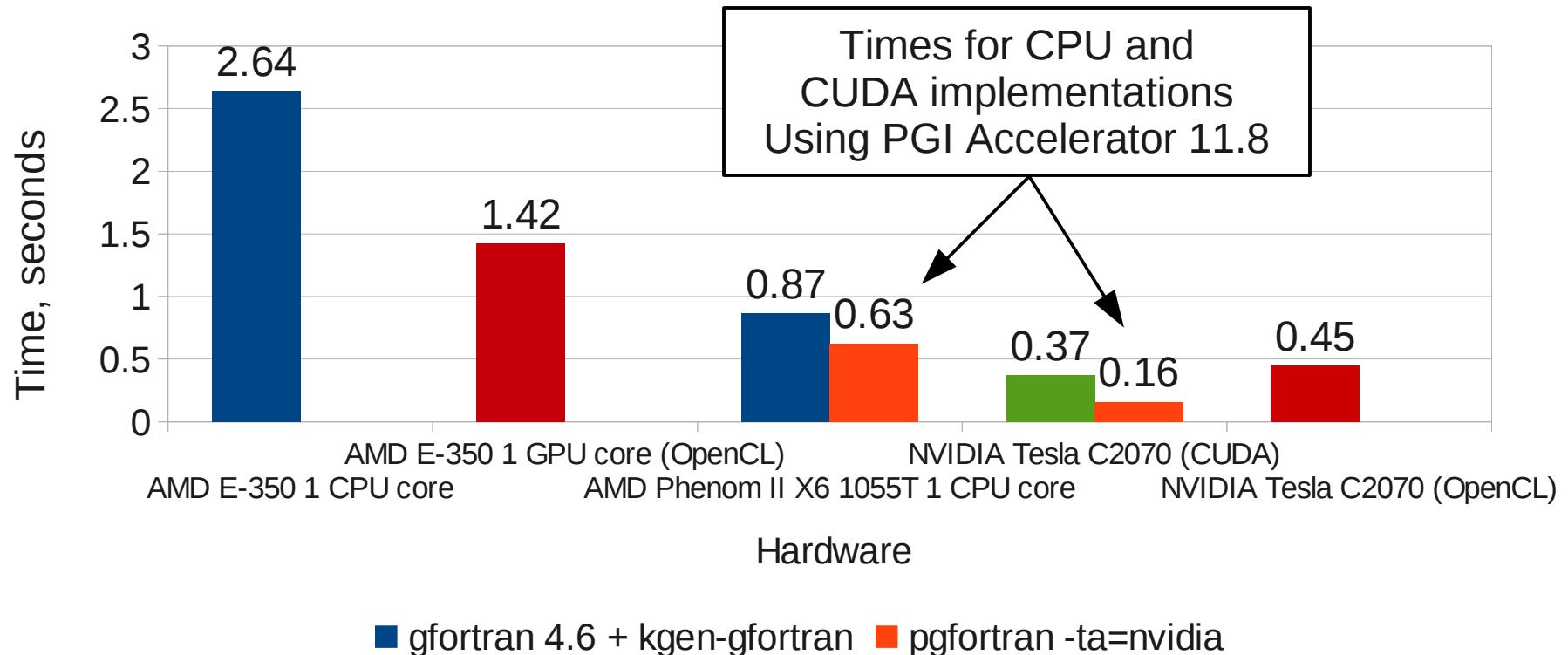
Example: sincos - performance

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

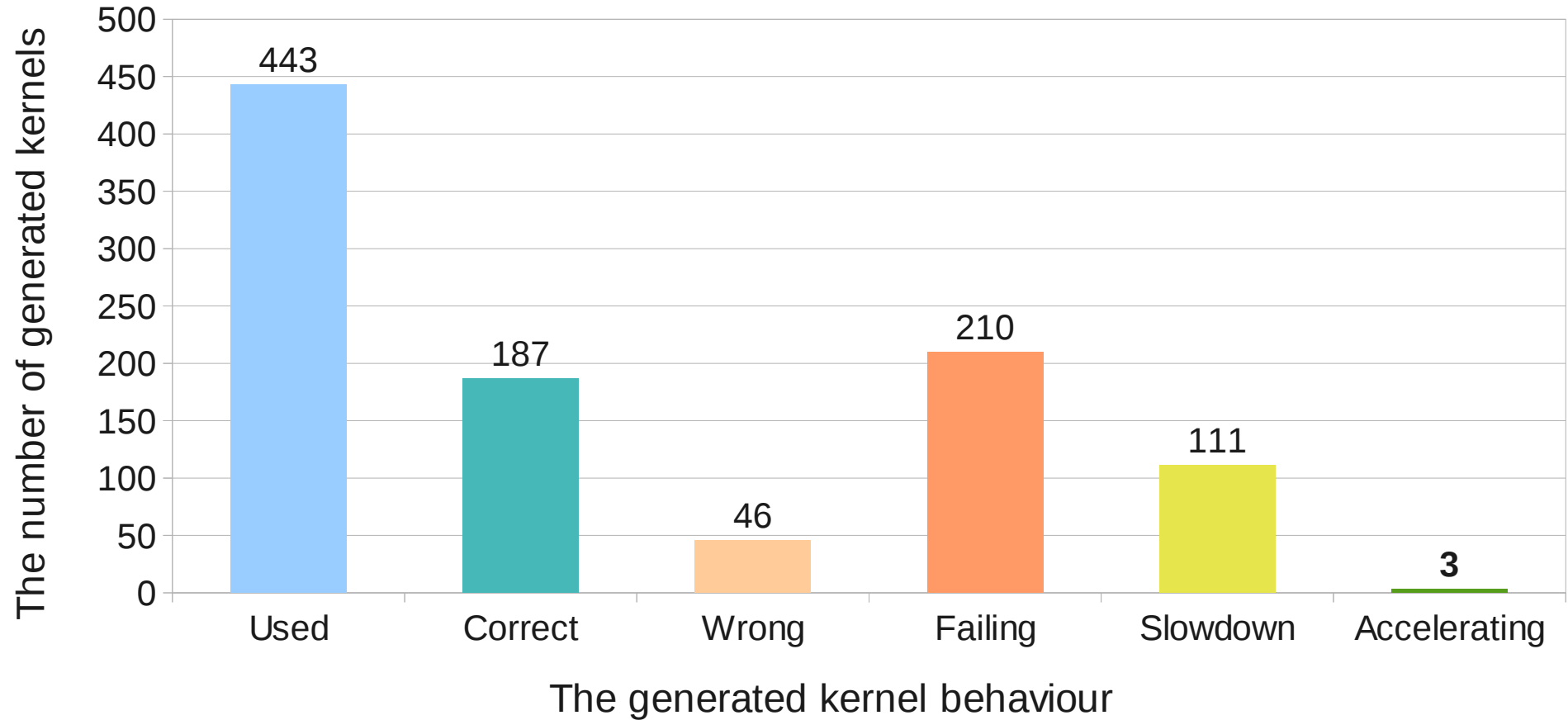


Example: sincos - performance

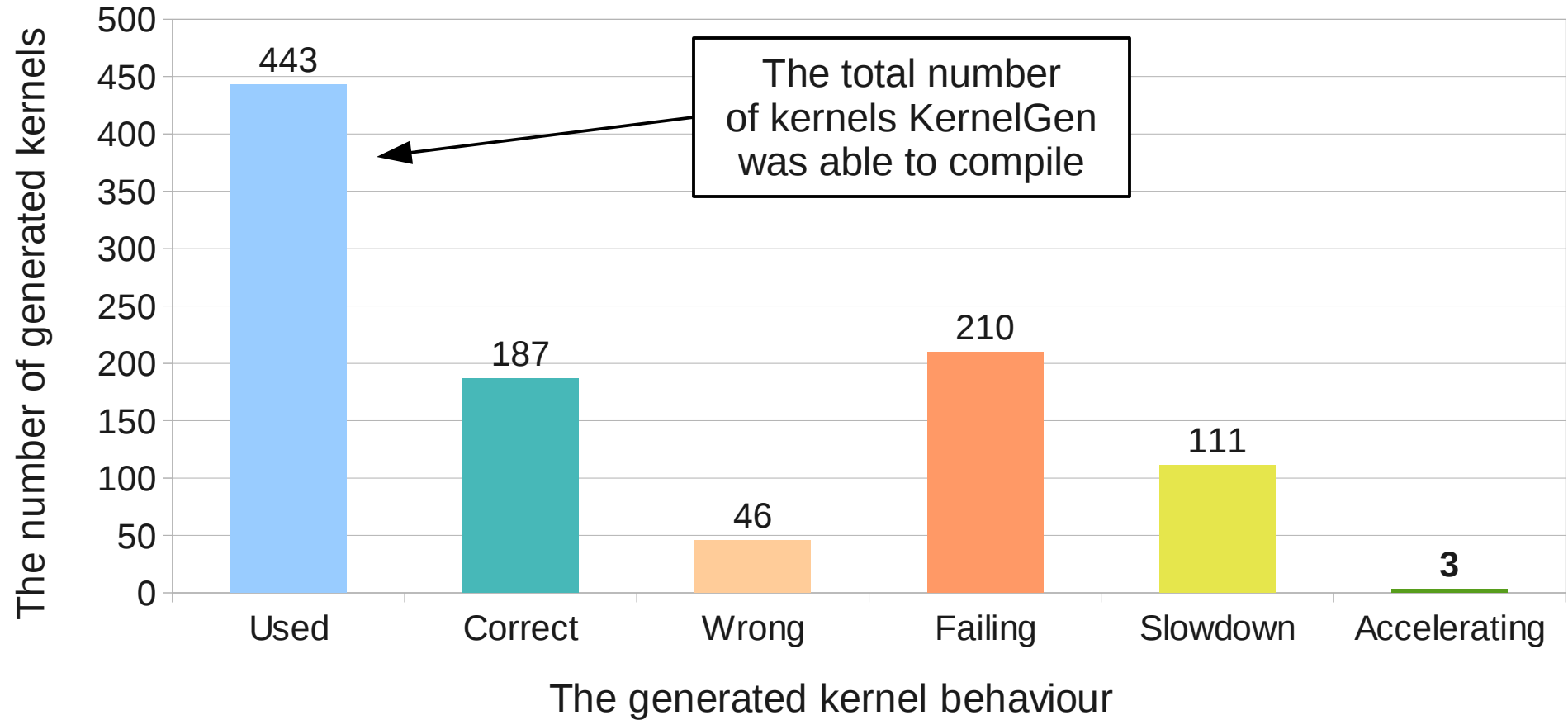
Performance of CPU binary generated by gfortran and OpenCL/CUDA kernels by KernelGen, compared to host and device perfs using PGI Accelerator 11.8 (orange)



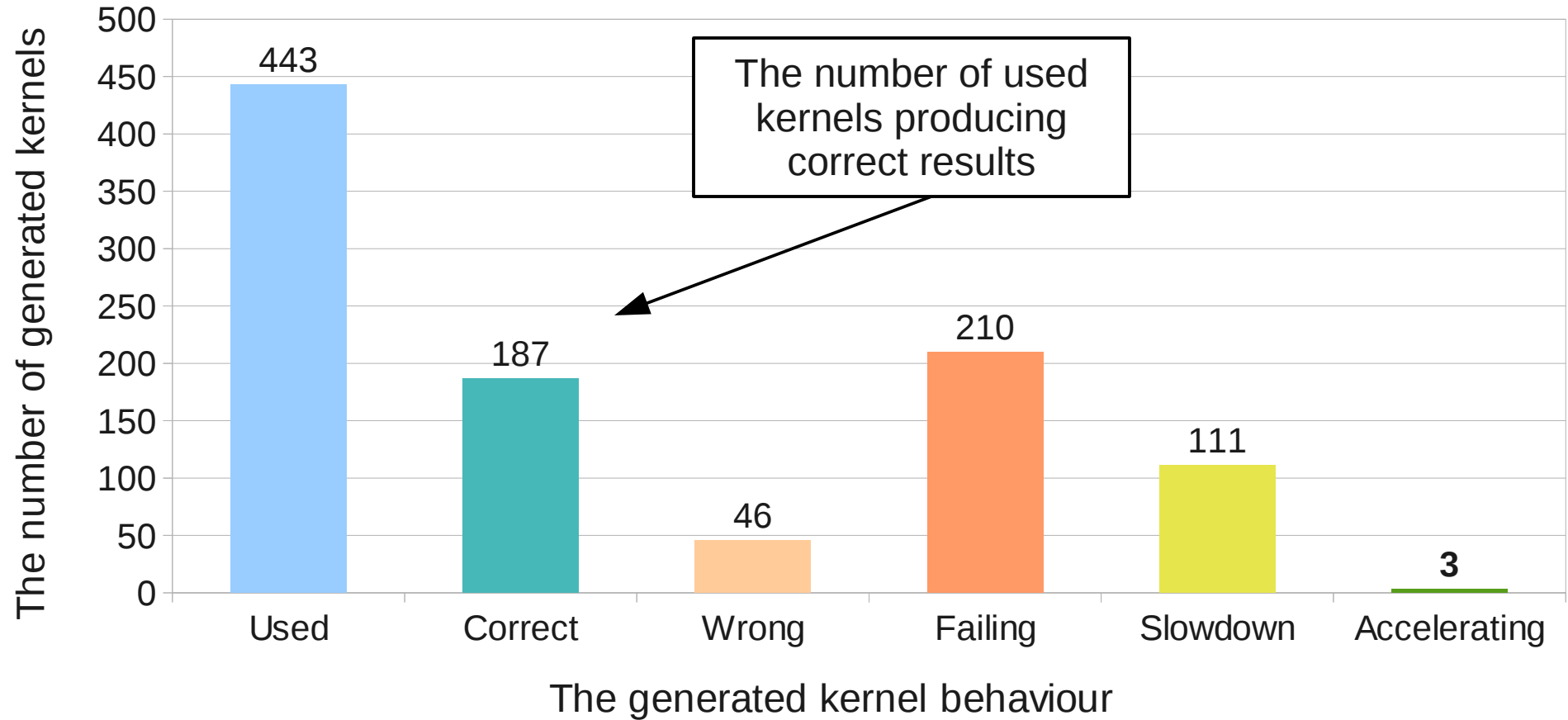
COSMO - coverage



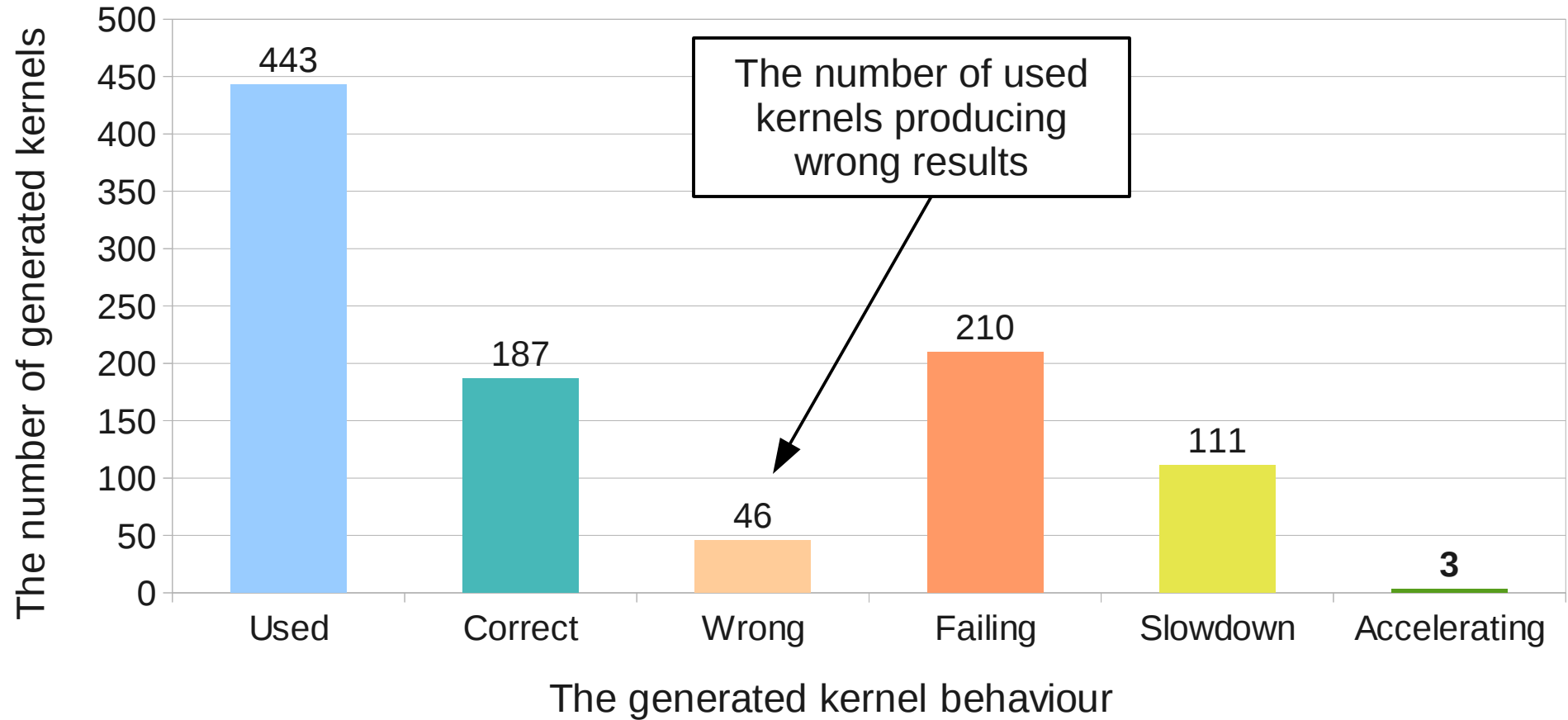
COSMO - coverage



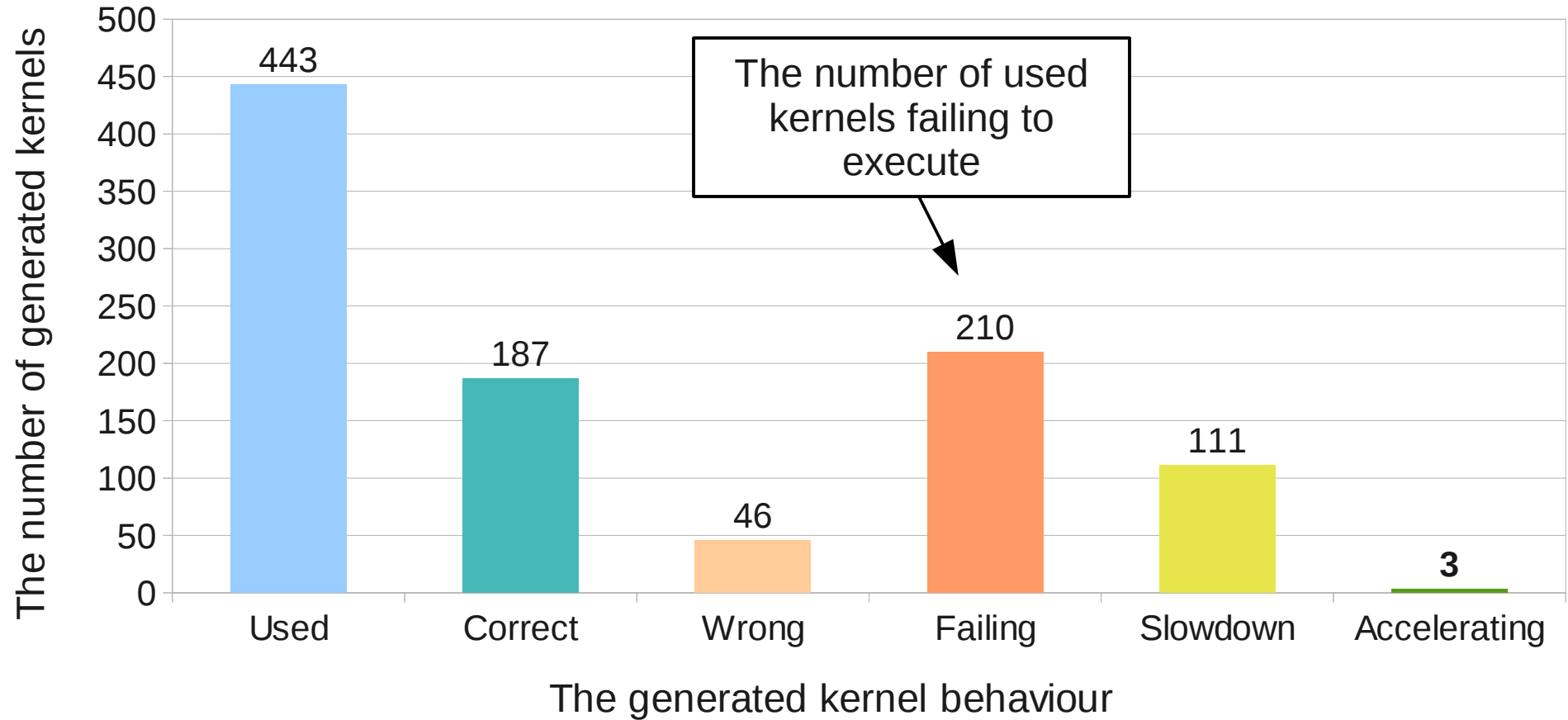
COSMO - coverage



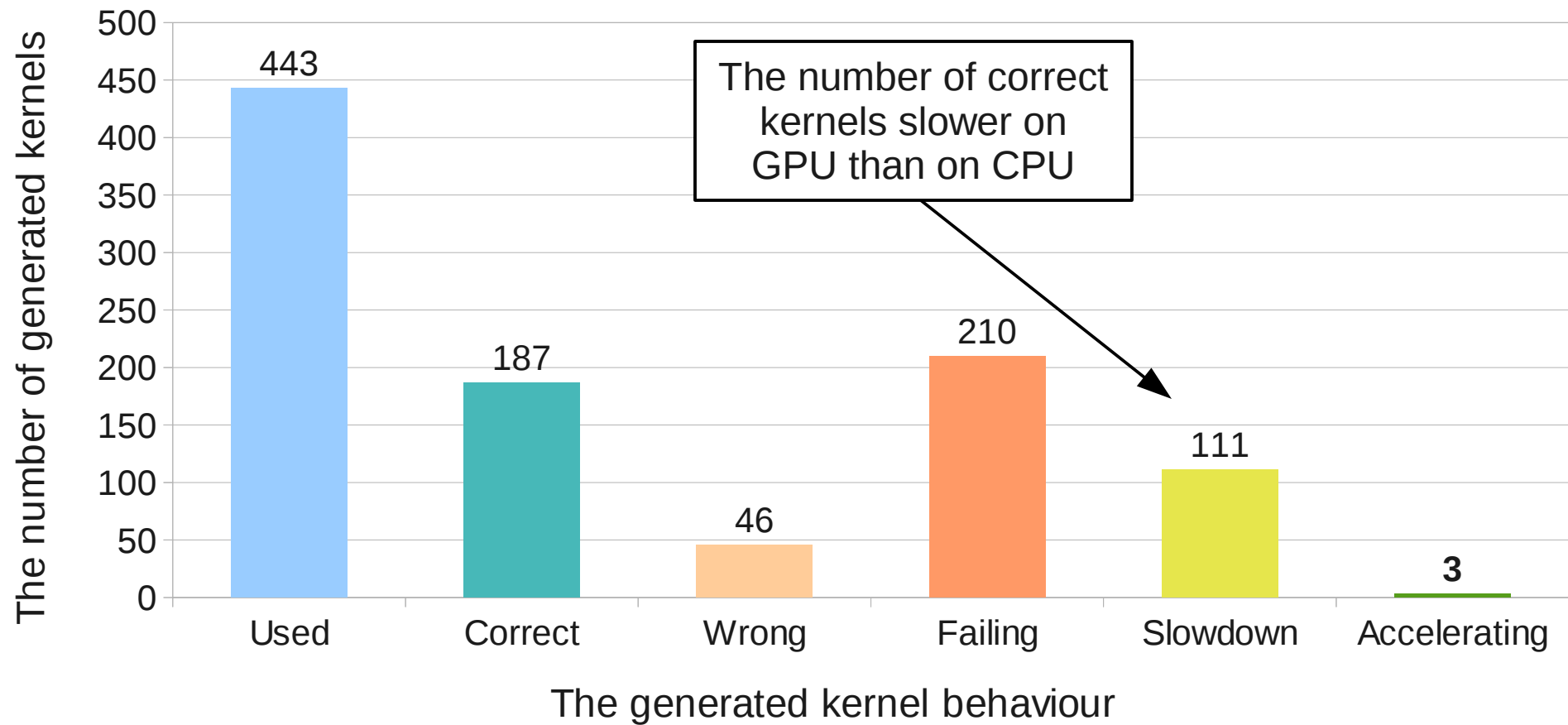
COSMO - coverage



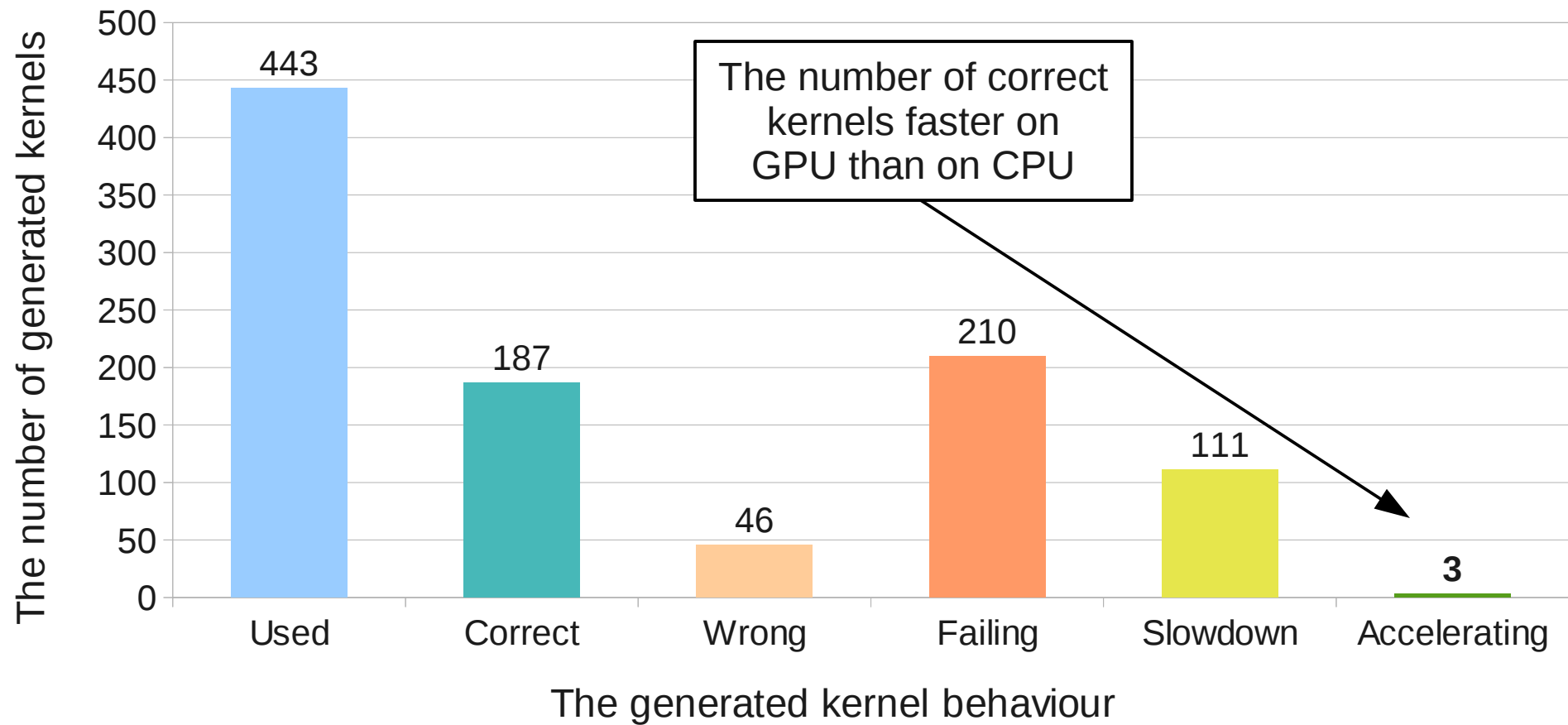
COSMO - coverage



COSMO - coverage

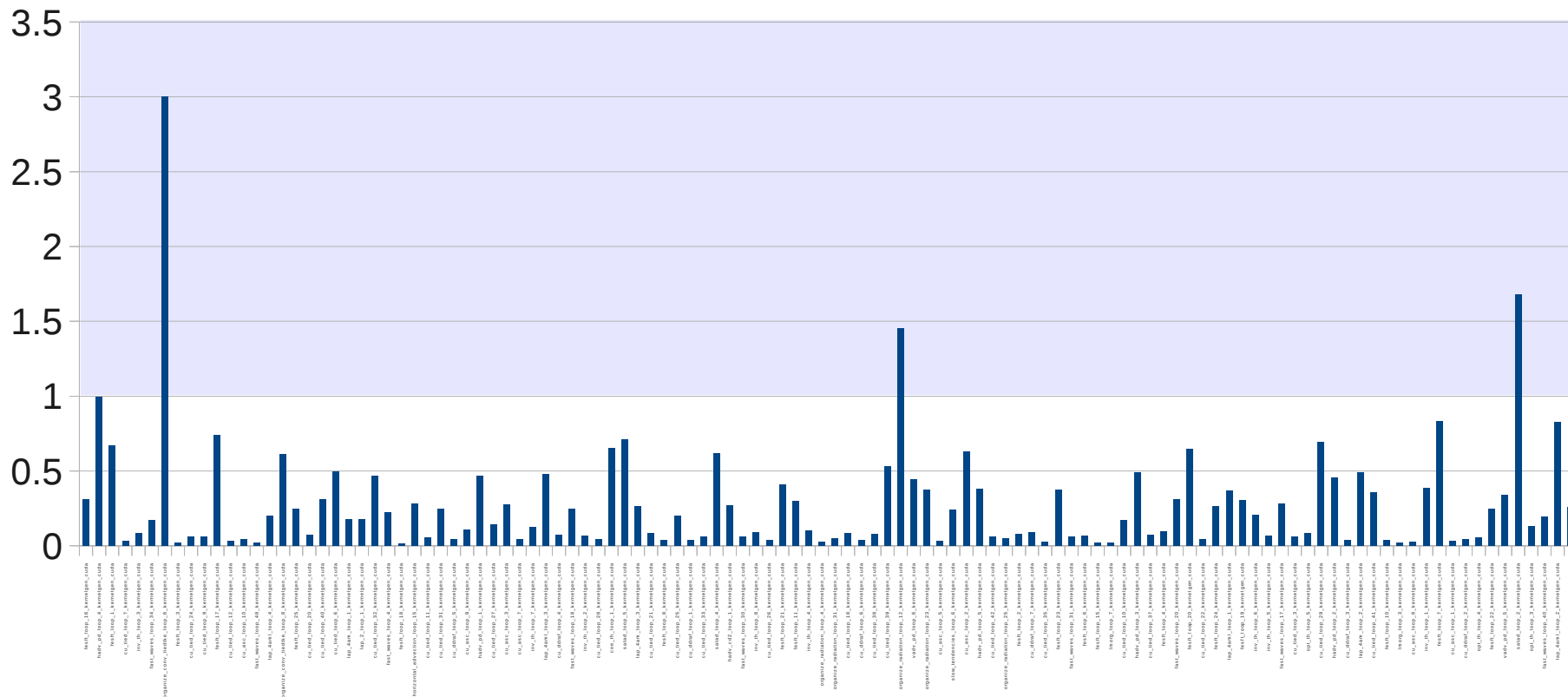


COSMO - coverage



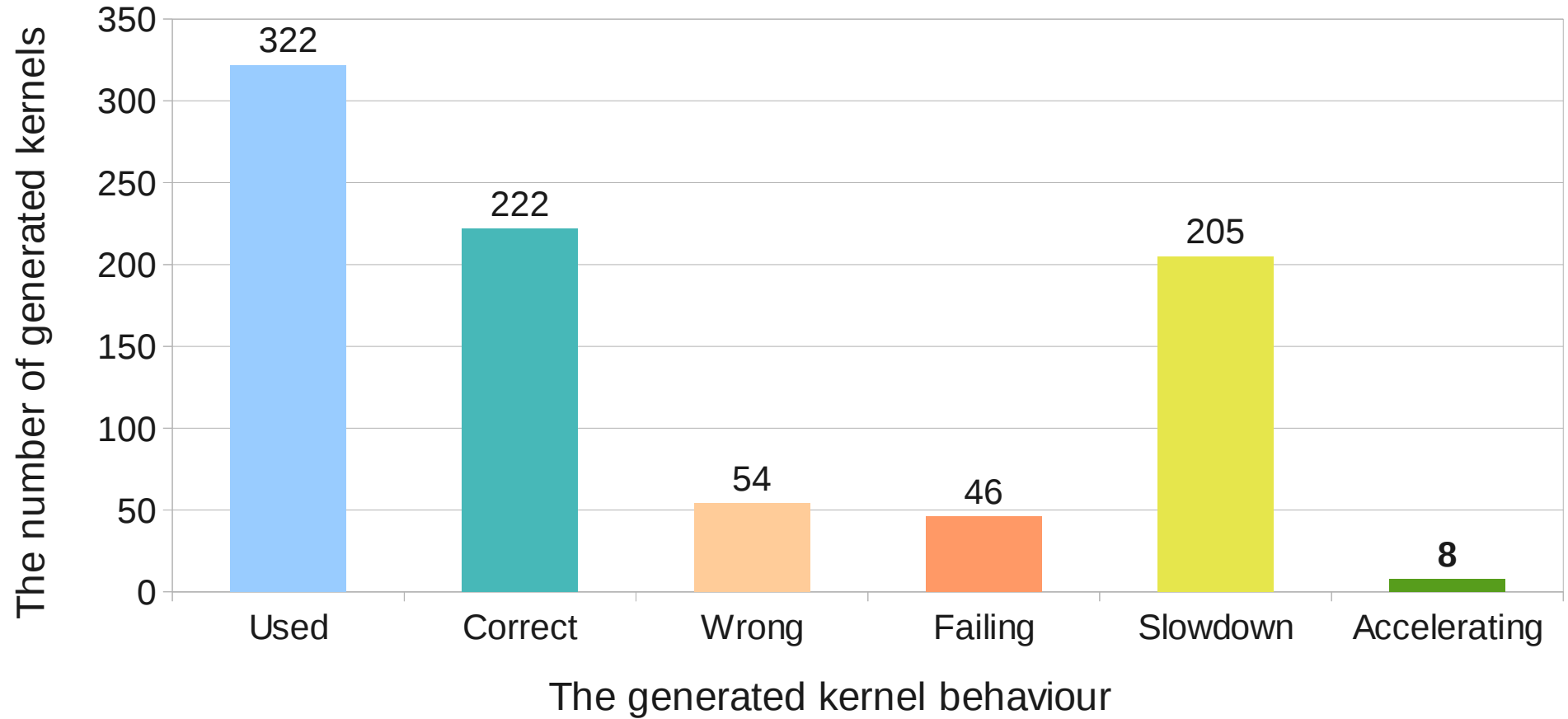
COSMO - performance

The generated kernel speedup, times

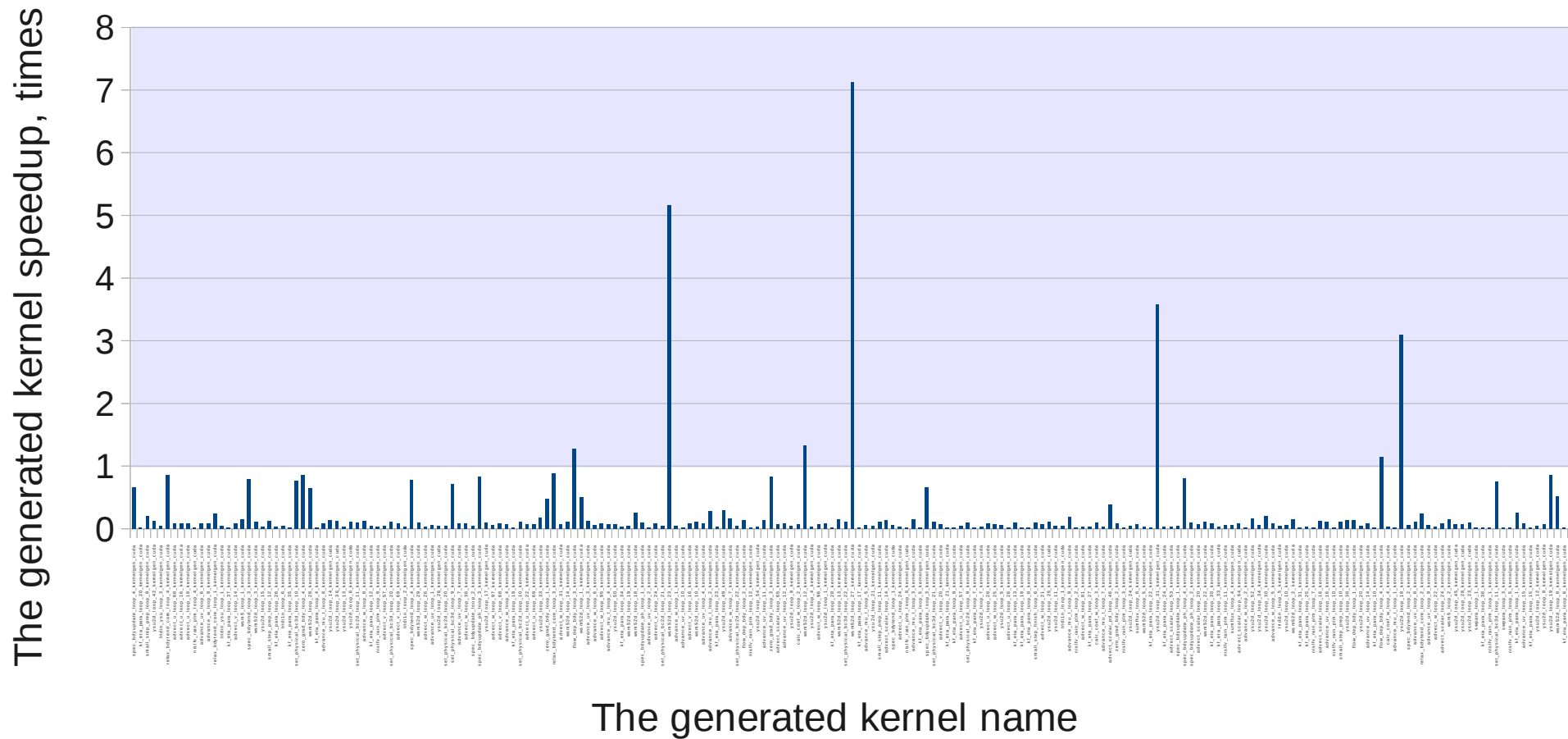


The generated kernel name

WRF - coverage

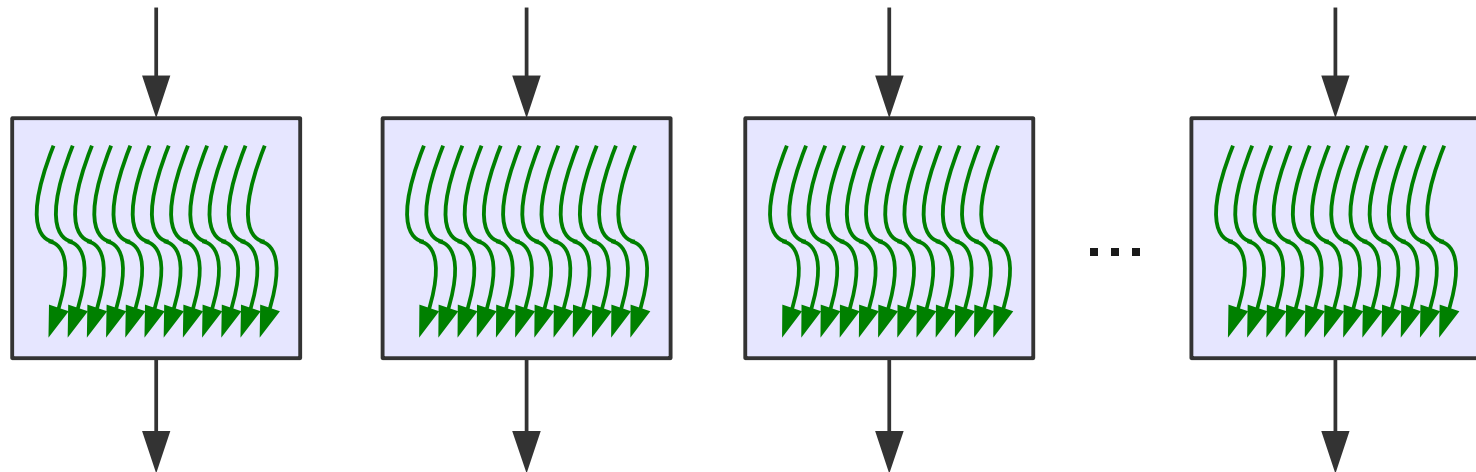


WRF - performance



Why slowdown?

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



5. 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 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

Stage 2 (July - October)

- Improve support/coverage  – done  – in progress now
 - **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

6. Team & resources

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 \Rightarrow 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](#)

Collaboration

We provide:

- Source code
- 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?