Introduction To GPU Programming

Martin Schwinzerl, Riccardo de Maria

HSS Section Meeting, CERN

June 3rd, 2020

Goals

- Overview about the GPU computing landscape (Concepts, Hardware)
- Introduce the (currently) two most established frameworks
- Provide Simple Examples to get started
- Outline the typical workflow within a GPU enhanced Program
- Performance Analysis& Constraints

- **Motivate Design-Decisions & Implementation of SixTrackLib**

Note: This is an interactive Jupyter Notebook - all presented examples are designed to work and allow you to experiment with them. Notbook available from: github.com (https://github.com/martinschwinzerl/sixtracklib-presentations/tree/master/20200603_hss_section_meeting)

GPU Hardware Categories

- **Low-End Gaming: ≈ 100 EUR**
  - ▷ Still equal or better FP performance than typical CPU in SP and DP
  - ▷ DP performance usualy much poorer than SP!

- AMD RX 560 TI 4 GByte
  - ▷ 1024 Cores
  - ▷ SP Peak Performance 2406 GFLOP/s
  - ▷ DP Peak Performance 163 GFLOP/s ≈ 1/16 SP
- NVidia GTX 1050 Ti 4 GByte
  - ▷ 768 Cores
  - ▷ SP Peak Performance 1981 GFLOP/s
  - ▷ DP Peak Performance 62 GFLOP/s = 1/32 SP

- SP ... IEEE754 32Bit single precision floating point numbers
- DP ... IEEE754 64Bit double precision floating point numbers

GPU Hardware Categories

- Low-End Gaming: ≈ 100 EUR
- **High-End Gaming: ≤ 1000 EUR**
  - ▷ Substancial performance in SP
  - ▷ DP performance varies!

- NVidia GeForce RTX 2080 8GByte
  - ▷ 3072 Cores
  - ▷ SP Peak Performance 8920 GFLOP/s
  - ▷ DP Peak Performance 279 GFLOP/s = 1/32 SP
- AMD RX Vega 64 8 GByte
  - ▷ 3840 Cores
  - ▷ SP Peak Performance 11518 GFLOP/s
  - ▷ DP Peak Performance 720 GFLOP/s = 1/16 SP

- SP ... IEEE754 32Bit single precision floating point numbers
- DP ... IEEE754 64Bit double precision floating point numbers

GPU Hardware Categories

- Low-End Gaming: ≈ 100 EUR
- High-End Gaming: ≤ 1 kEUR
- **Hybrid HPC: ≈ 3 kEUR - 8 kEUR**
  - ▷ Substantial performance in SP
  - ▷ DP performance ≈ 1/2 SP
- **Server HPC: ≈ 6 kEUR - 8 kEUR**
  - ▷ Similar performance characteristics as with hybrid HPC, but
  - ▷ No video outputs, i.e. "Accelerator Card"

- NVidia Titan V 12 GBytes
  - ▷ 3072 Cores
  - ▷ Hybrid card - Video outputs available
  - ▷ SP Peak Performance 12288 GFLOP/s
  - ▷ DP Peak Performance 6144 GFLOP/s = 1/2 SP
- AMD Radeon Instinct MI50 16 GBytes
  - ▷ 3840 Cores
  - ▷ SP Peak Performance 13400 GFLOP/s
  - ▷ DP Peak Performance 6700 GFLOP/s = 1/2 SP

- SP ... IEEE754 32Bit single precision floating point numbers
- DP ... IEEE754 64Bit double precision floating point numbers

GPU Hardware Categories

- Low-End Gaming: ≈ 100 EUR
- High-End Gaming: ≤ 1 kEUR
- Hybrid HPC: ≈ 3 kEUR - 8 kEUR
- Server HPC: ≈ 6 kEUR - 8 kEUR
- **AI Server HPC: ≈ 6 kEUR - 10 kEUR**

- Substancial Integer, SP and Half-Precision Performance or
- TOPS ... Trillion operations per second of dedicated neural network computes
- Task-specific Hybrid systems with FPGA and/or CPU enhancements
- No classical GPU, no video outputs "Accelerator Card"

</div>

- SP ... IEEE754 32Bit single precision floating point numbers
- DP ... IEEE754 64Bit double precision floating point numbers

CPUs versus GPUs

	CPU	GPU
Cores / Compute Units	2 - 64 (128 HT/SMT)	16 - 80
Arithmetic Units / Core	2-8	64
Peak Performance SP	24-3200 GFLOP/s	1000-19500 GFLOP/s
Peak Performance DP	12-1600 GFLOP/s	50-9700 GFLOP/s

Compared to CPUs, GPUs have:

- More arithmetic units (in particular SP)
- Less logic for control flow
- Less registers per arithmetic units
- Less memory but larger memory bandwidth

</div>

- SP ... IEEE754 32Bit single precision floating point numbers
- DP ... IEEE754 64Bit double precision floating point numbers
- For CPUs: DP performance is roughly 1/2 of SP performance (YMMV)

Introducing Parallelism

- **Task Parallelism:** perform multiple tasks on the same dataset

Example: Check if a number is a prime number by checking divisibility in parallel

Introducing Parallelism

- **Data Parallelism:** perform the same task on multiple datasets

Example: Vector addition

$$
\begin{array}{ccccc}
\vec{z} & = & \vec{x} & + & \vec{y} \\
\boxed{z_0} & = & \boxed{x_0} & + & \boxed{y_0} \\
\boxed{z_1} & = & \boxed{x_1} & + & \boxed{y_1} \\
\boxed{z_2} & = & \boxed{x_2} & + & \boxed{y_2} \\
\boxed{z_3} & = & \boxed{x_3} & + & \boxed{y_3} \\
\vdots & & \vdots & & \vdots \\
\boxed{z_i} & = & \boxed{x_i} & + & \boxed{y_i} \\
\vdots & & \vdots & & \vdots \\
\boxed{z_{n-1}} & = & \boxed{x_{n-1}} & + & \boxed{y_{n-1}}
\end{array}
$$

Data Parallelism:

Perform a

Single Instruction on Multiple Datasets

⇨ SIMD

or a

Single Program on Multiple Datasets

⇨ SPMD

$\vec{x}\,, \vec{y}\,, \vec{z} \in \mathbb{R}^n$

Example: Vector Addtion on the GPU using CuPy

Cf. https://github.com/cupy/cupy (https://github.com/cupy/cupy) and https://cupy.chainer.org/ (https://cupy.chainer.org/) for reference

In [1]:

```
# If you have a NVidia GPU, one of the simplest ways to offload calculations to the GPU is to use CuPy

import cupy
import numpy as np

N = 10000000 # 1.0e7 elements

x_host = np.random.rand( N )
y_host = np.random.rand( N )

# Create vectors of random numbers. CuPy handles all the required steps behind the scenes
x = cupy.asarray( x_host )
y = cupy.asarray( y_host )

# x and y are cupy entities that "live" on the GPU while
# x_host and y_host "live" in regular memory
# The same applies to the result of the calculation z:

z = x + y  # Vector addition is actually performed on the GPU
```

In [2]:

```
# in order to make use of the result outside of the GPU,
# we have to convert z into a regular numpy array
z_host = cupy.asnumpy( z )

# Compare result to calculation on CPU
print( f"calculation on host and device yield same result: {np.allclose( z_host, x_host + y_host, rtol=0.0, atol=1e-16)}" )

del x, y, z, x_host, y_host, z_host
```

```
calculation on host and device yield same result: True
```

Overview: Frameworks, Libraries, Toolboxes for GPU Programming

- High-Level Libraries and Applications
  - ▷ CuPy: numpy-like python module for NVIDIA GPUs
  - ▷ Tensorflow: mostly support for NVIDIA GPUs, some opportunities to use other cards via SYCL
  - ▷ Matlab (currently only NVIDIA GPUs are supported)
  - ▷ Julia
  - ▷ Mathematica
  - ▷ ....
- HPC frameworks & libraries
  - ▷ OpenMP: ≥ version 4.5, allows offloading to GPU targets (for AMD, a recent gcc or clang is required)
  - ▷ OpenACC: similar to OpenMP. Traditionally NVIDIA focused but recently got capabilities to offload to AMD
  - ▷ Intel OneAPI
  - ▷ ArrayFire: optimized functions and data-structures to exploit parallelism on both GPUs and CPUs

Overview: Frameworks, Libraries, Toolboxes for GPU Programming

- Low-Level Programming Frameworks
  - ▷ ROCm: collection of libraries & tools for AMD GPUs, provides OpenCL backend
  - ▷ HIP: abstract language to write GPU programs (AMD, open-source), allows "automated" translation of CUDA programs into HIP
  - ▷ SyCL: "modernized" abstraction on-top of OpenCL, uses SPIR
  - ▷ clang + SPIR-V: compile code fragments to a intermediate format consumable by modern OpenCL implementations
  - ▷ clang + PTX: compile code fragments to an intermediate format consumable by CUDA (NVRTC)
  - ▷ CUDA C99/C++1x based framework for NVIDIA GPUs (some parts are open-source, drivers etc. are proprietary)
  - ▷ OpenCL: Vendor neutral computing language, allows to target CPUs, GPUs, FPGAs, Signal Processors

- Increasingly Important: Code-Transformation tools and intermediate code representations
  - ▷ PTX: Parallel Thread Execution language, used by NVIDIA for CUDA
  - ▷ SPIR, SPIR-V: Standard Portable Intermediate Representation, cross-platform open standard maintained by the Khronos Group; Central in newer graphic standards and newer iterations of OpenCL

Comparison: OpenCL versus CUDA

- OpenCL
  - Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
  - Common, standardized software stack (Khronos Group) + vendor specific impl.
  - "Programming to the least common denominator"
  - OpenCL 1.2: Still most commonly used, C99 kernel language, limited set of features
  - OpenCL 2.x: some support from AMD and Intel
  - OpenCL 3.0: recently specified, modular refactoring of OpenCL 2.x

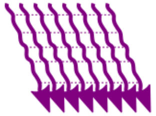
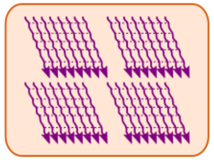
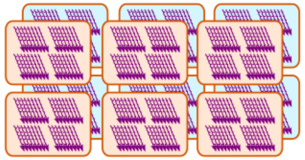
- CUDA
  - Hardware: Only NVIDIA GPUs, no CPU / emulator available
  - Software implementation only available from NVIDIA, includes libraries, debuggers, profilers, memory checkcers, etc.
  - Software and hardware tightly integrated (versioning, "compute capabilites")
  - Version 10.x allowing both C99 and C++11/14 kernel language

- **Primarily**: Run-time compilation of the kernels
- Kernel code has to be available at run time, compile for selected device at run-time
- Single-file compilation similar to the default in CUDA available (SyCL)

- **Primarily**: Single-file compilation. nvcc compiler replaces sysetem compiler for .cu files -> binary compability between system compiler, system libraries, graphic stack, etc. required
- Run-time compilation approach simliar to the default with OpenCL is available (NVRTC)

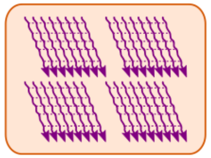
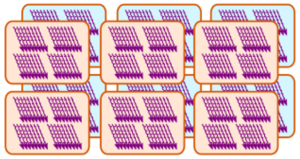
CUDA and OpenCL: Different But Similar - Grid

- Threads are organized in two layers resembling a 3D grid
- Simpler use-cases only requiring 1D or 2D topologies are mapped on the 3D structure

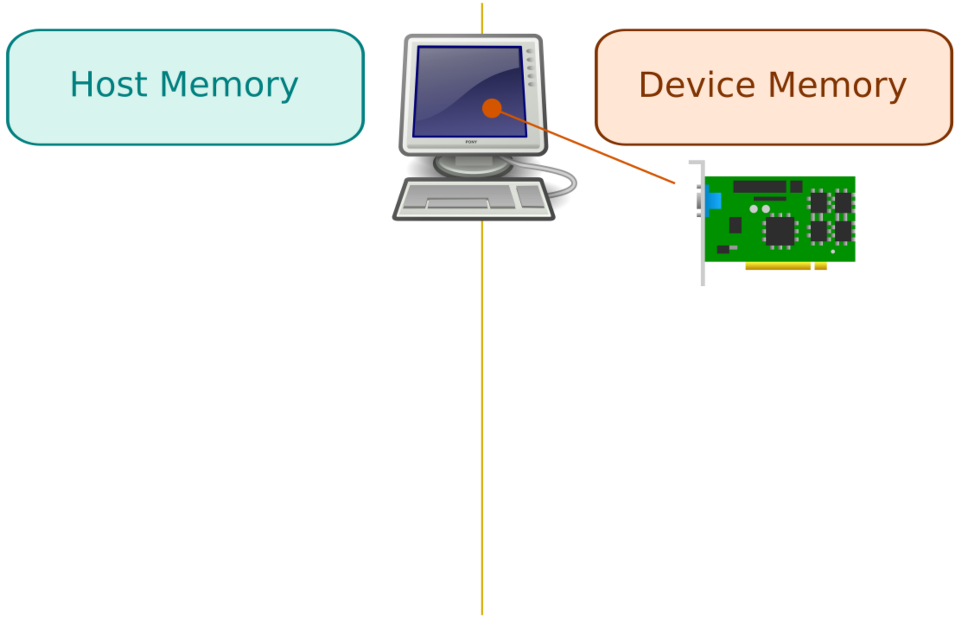
	OpenCL	CUDA
	Workitem	Thread
	AMD: Wavefront NVIDIA: Warp Vectorized CPU Code: Datapath Width	Warp
	Work-Group	Block
	NDRange	Grid

CUDA and OpenCL: Different But Similar - Memory Model

- Hierarchical memory model with different "regions"
- Visibility allows data exchange across individual threads / work-items but may require synchronization
- On CUDA, closely linked to Hardware implementation
- On OpenCL, guarantees about different memory regions are more difficult to give

	OpenCL	CUDA	
	__private or no keyword	no keyword	available locally in thread / workitem
	__local	__shared__	available to all threads / workitems in a block / workgroup
	__global	__device__	available to all threads / workitems in grid / ndrange
	__constant	__device__ __constant__	

Structure Of A Simple GPU Program

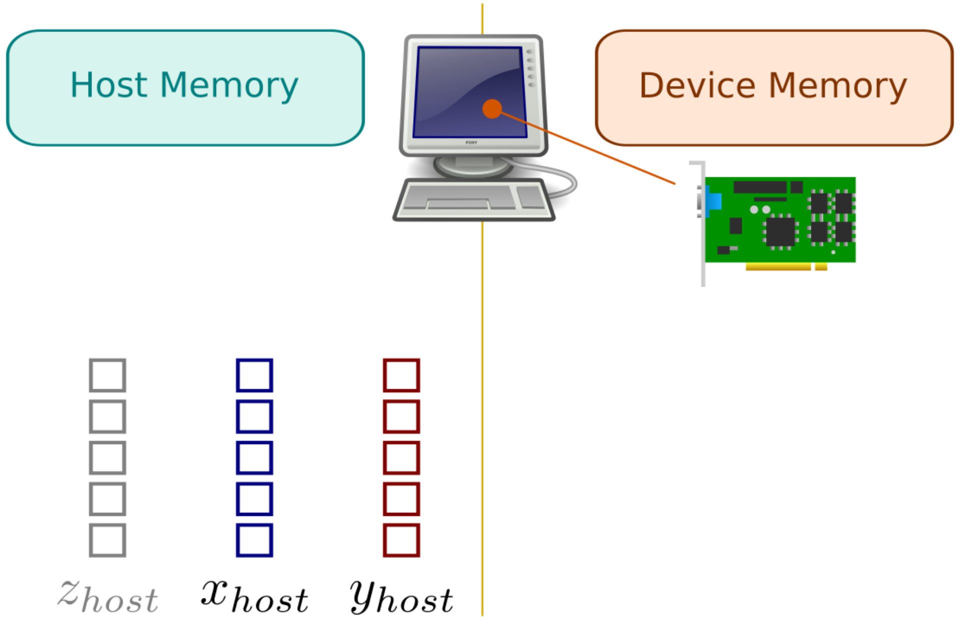


Host Memory

Device Memory

Icons: https://openclipart.org - License: Public Domain

Structure Of A Simple GPU Program



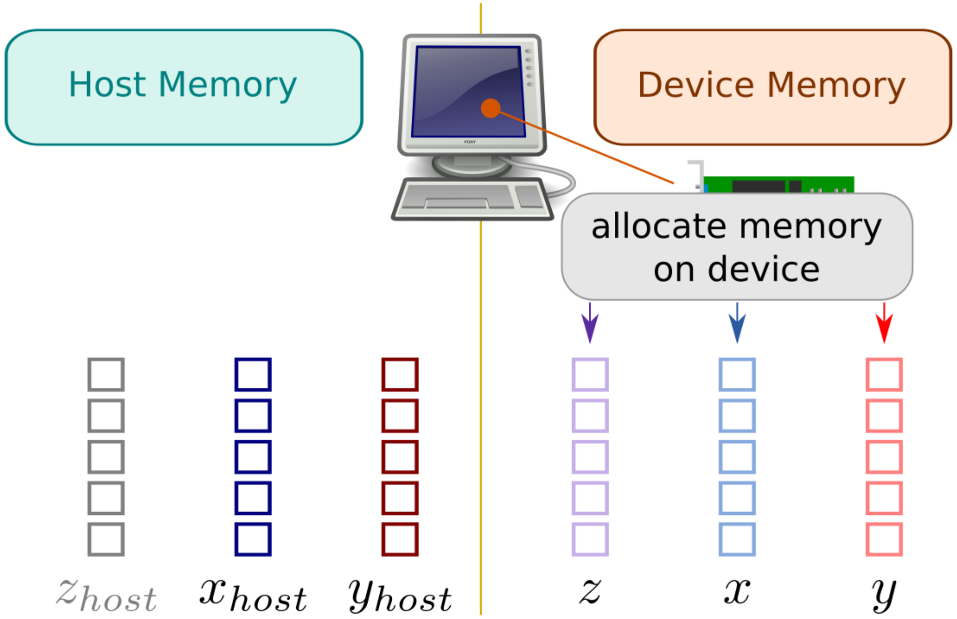
Host Memory

Device Memory

z_{host} x_{host} y_{host}

Icons: https://openclipart.org - License: Public Domain

Structure Of A Simple GPU Program



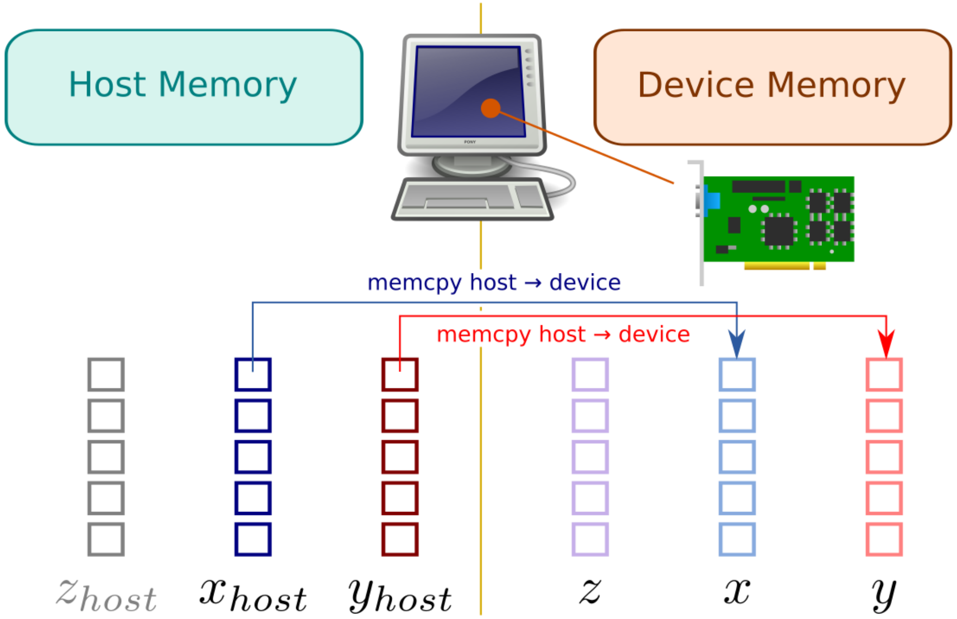
Host Memory

Device Memory

allocate memory on device

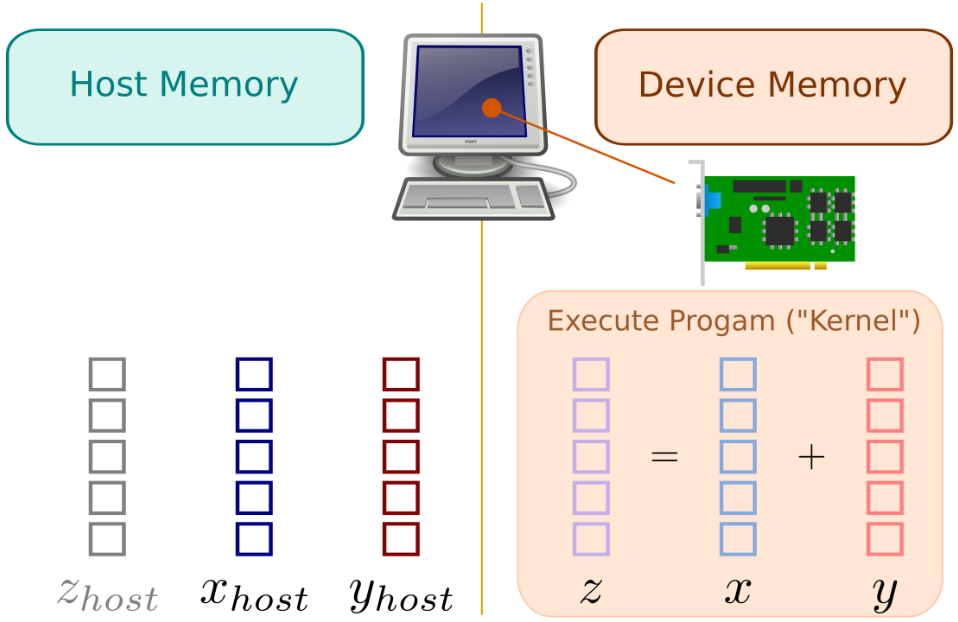
Icons: https://openclipart.org - License: Public Domain

Structure Of A Simple GPU Program



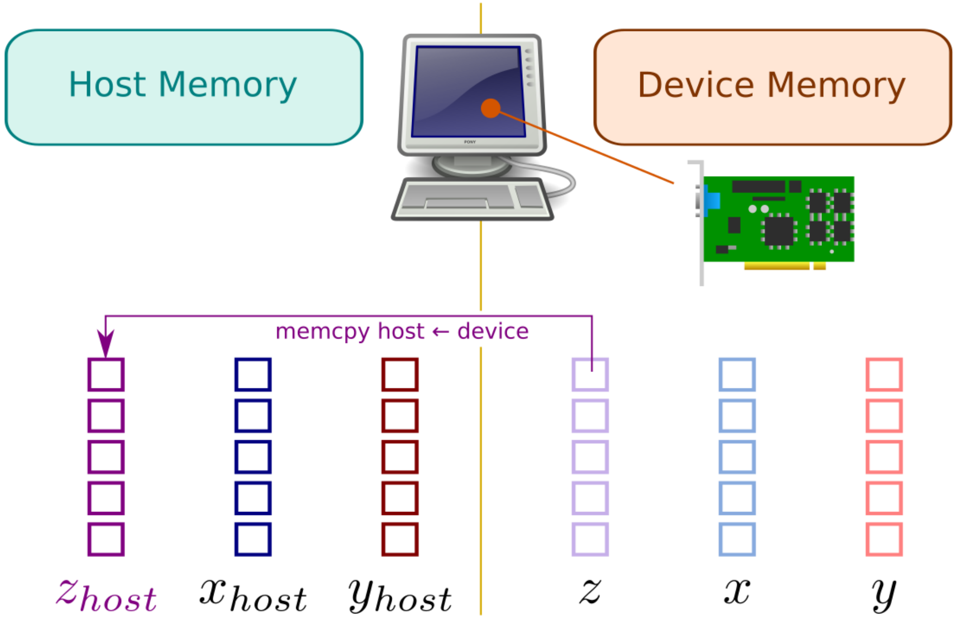
Icons: https://openclipart.org - License: Public Domain

Structure Of A Simple GPU Program



Icons: https://openclipart.org - License: Public Domain

Structure Of A Simple GPU Program



Host Memory

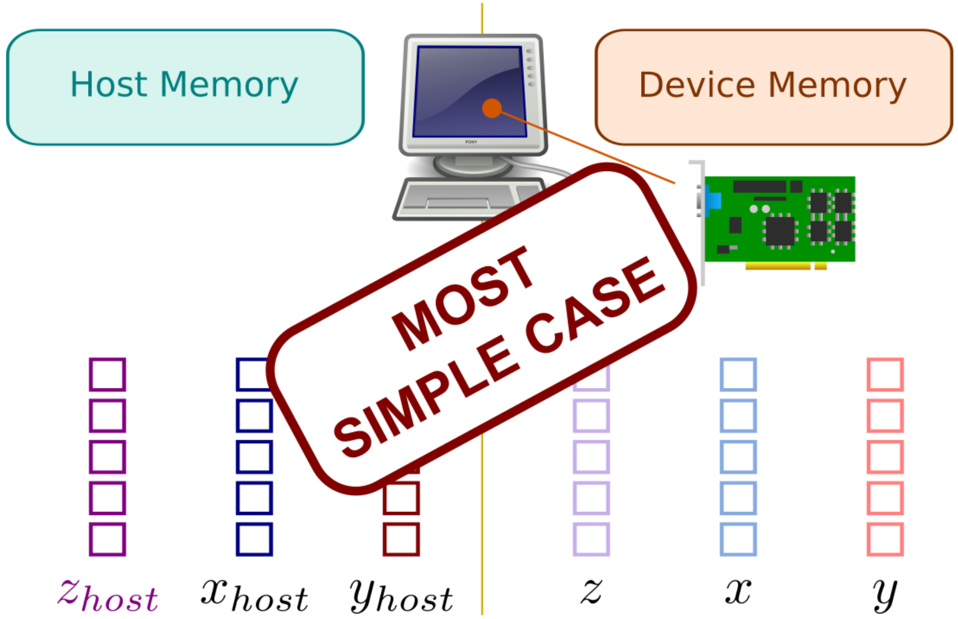
Device Memory

memcpy host ← device

z_{host} x_{host} y_{host} z x y

Icons: https://openclipart.org - License: Public Domain

Structure Of A Simple GPU Program



Icons: https://openclipart.org - License: Public Domain

Example: Vector Addtion on the GPU using PyCuda

Cf. https://github.com/inducer/pycuda (https://github.com/inducer/pycuda) and https://documen.tician.de/pycuda/ (https://documen.tician.de/pycuda/) for reference

In [3]:

```
# Again, a NVIDIA GPU is required for this example to work!
import pycuda.autoinit
import pycuda.driver as cuda
import numpy as np
from pycuda.compiler import SourceModule

# First, we prepare the program that should be applied to our data, i.e. the "K
ernel"
vec_add_program = SourceModule(
"""
__global__ void sum_kernel( double* __restrict__ z,
                            double const* __restrict__ x,
                            double const* __restrict__ y, int const N )
{
    const int threads_per_block = blockDim.x; /* Assuming 1D grid */
    const int ii = threadIdx.x + blockIdx.x * threads_per_block; /* Assuming 1D
grid */
    if( ii < N )
    {
        z[ ii ] = x[ ii ] + y[ ii ];
    }
}
""")

# Note: even though we are using CUDA, we are able to specify the kernel at run
-time ->
# PyCuda uses the NVRTC implementation!

vec_add_kernel = vec_add_program.get_function( "sum_kernel" )
```

In [4]:

```
# then, we prepare the structures on the host side, allocate the
# required resources on the device side and move things

N = 10000000 # Again, 10^7 elements to the vector

x_host = np.random.randn( N ) # prepare the elements on the host
y_host = np.random.randn( N )

# allocate the structures on the device
x = cuda.mem_alloc( x_host.nbytes )
y = cuda.mem_alloc( y_host.nbytes )
z = cuda.mem_alloc( max( x_host.nbytes, y_host.nbytes ) )

#transfer the memory from the host to the device; htod ... host to device
cuda.memcpy_htod( x, x_host )
cuda.memcpy_htod( y, y_host )
```

In [5]:

```python
# find the grid dimensions:
threads_per_block = 128 # This is a pessimistic but educated guess :-)
num_blocks = N // threads_per_block
if N % threads_per_block != 0:
    num_blocks += 1

# run the kernel
N_arg = np.int32(N) # pycuda enforces strict type checking
vec_add_kernel( z, x, y, np.int32(N), block=(threads_per_block, 1, 1), grid=(num_blocks,1))

# copy the result back to the host; dtoh ... device to host
z_host = np.empty_like( x_host )
cuda.memcpy_dtoh( z_host, z )


# Compare result to calculation on CPU
print( f"calculation on host and device yield same result: {np.allclose( z_host, x_host + y_host, rtol=0.0, atol=1e-16)}" )

del x, y, z, x_host, y_host, z_host
```

```
calculation on host and device yield same result: True
```

Example: Vector Addtion on the GPU using PyOpenCL

Cf. https://github.com/inducer/pyopencl (https://github.com/inducer/pycuda) and https://https://documen.tician.de/pyopencl/ (https://documen.tician.de/pyopencl/) for reference

In [6]:

```
import pyopencl as cl
import numpy as np

# setup the OpenCL environment - it's a bit more elaborated as with PyCuda:
# Shortcut: ctx = cl.create_some_context() -> create a context with the first a
vailable device

platforms = cl.get_platforms()
for p in platforms:
    print( p )
```

```
<pyopencl.Platform 'AMD Accelerated Parallel Processing' at 0x7fc00450a250>
<pyopencl.Platform 'NVIDIA CUDA' at 0x3d31de0>
<pyopencl.Platform 'Portable Computing Language' at 0x7fc00dece020>
```

In [9]:

```
platform = platforms[ 0 ] # Let's select the first one

# get a list of all available devices for a platform
devices = platform.get_devices()

for d in devices:
    print( d )
```

```
<pyopencl.Device 'gfx900' on 'AMD Accelerated Parallel Processing' at 0x33ec26
0>
```

In [10]:

```
# Again, let's select the first device
device = devices[ 0 ]

# create a context and a queue
ctx = cl.Context( [device,] )
queue = cl.CommandQueue( ctx )
```

In [11]:

```
# prepare the program containing the kernel:
vec_add_program = cl.Program( ctx, """
__kernel void sum_kernel( __global double* restrict z,
                          __global double const* restrict x,
                          __global double const* restrict y, int const N )
{
    int const ii = ( int )get_global_id( 0 ); /* Assume 1D Grid, 0 .. 0th eleme
nt of ndrange */
    if( ii < N )
    {
        z[ ii ] = x[ ii ] + y[ ii ];
    }
}

""" )

# Compile the program containing the kernel
vec_add_program.build()

# get kernel function from vec_add_program
vec_add_kernel = vec_add_program.sum_kernel
```

In [12]:

```
# then, we prepare the structures on the host side, allocate the
# required resources on the device side and move things
N = 10000000 # Again, 10^7 elements to the vector

# create the structures on the host
x_host = np.random.rand( N )
y_host = np.random.rand( N )


# allocate the structures on the device
x = cl.Buffer( ctx, cl.mem_flags.READ_ONLY, x_host.nbytes )
y = cl.Buffer( ctx, cl.mem_flags.READ_ONLY, y_host.nbytes )
z = cl.Buffer( ctx, cl.mem_flags.WRITE_ONLY, max( x_host.nbytes, y_host.nbytes
) )

#transfer the memory from the host to the device
cl.enqueue_copy( queue, x, x_host )
cl.enqueue_copy( queue, y, y_host )
```

Out[12]:

```
<pyopencl._cl.NannyEvent at 0x7fc08f85b090>
```

In [13]:

```
# find the grid dimensions:
num_work_items = x_host.shape
workgroup_size = None # let the OpenCL impl find the optimal workgroup size

# run the kernel
Narg = np.int32(N) # like with OpenCL
vec_add_kernel( queue, num_work_items, workgroup_size, z, x, y, Narg )

# copy the result back to the host
z_host = np.empty_like( x_host )
cl.enqueue_copy( queue, z_host, z )
```

Out[13]:

```
<pyopencl._cl.NannyEvent at 0x7fc08f852360>
```

In [14]:

```
# Compare result to calculation on CPU
print( f"calculation on host and device yield same result: {np.allclose( z_host, x_host + y_host, rtol=0.0, atol=1e-16)}" )

del x, y, z, x_host, y_host, z_host
```

```
calculation on host and device yield same result: True
```

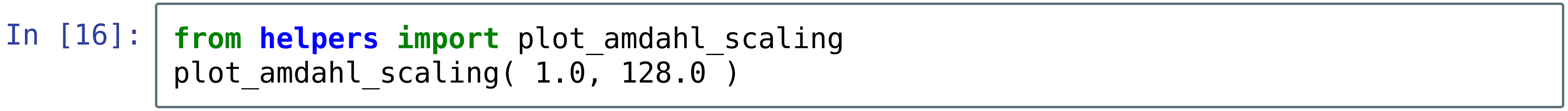
Performance Analysis: Scaling

- Program with a total run-time T
- Assumption: problem size is **constant** (e.g. add vectors of size N, track N particles, etc.)
- Question: how much can we speed up the execution of the problem if we parallelise it, i.e. the speed-up $\eta_s(n_p)$

- $T = t_p + t_s$
- t_p: fraction of run-time that can be run in "parallel" on n_p "processors"
- t_s: fraction of run-time that is sequential
- Amdahls law:

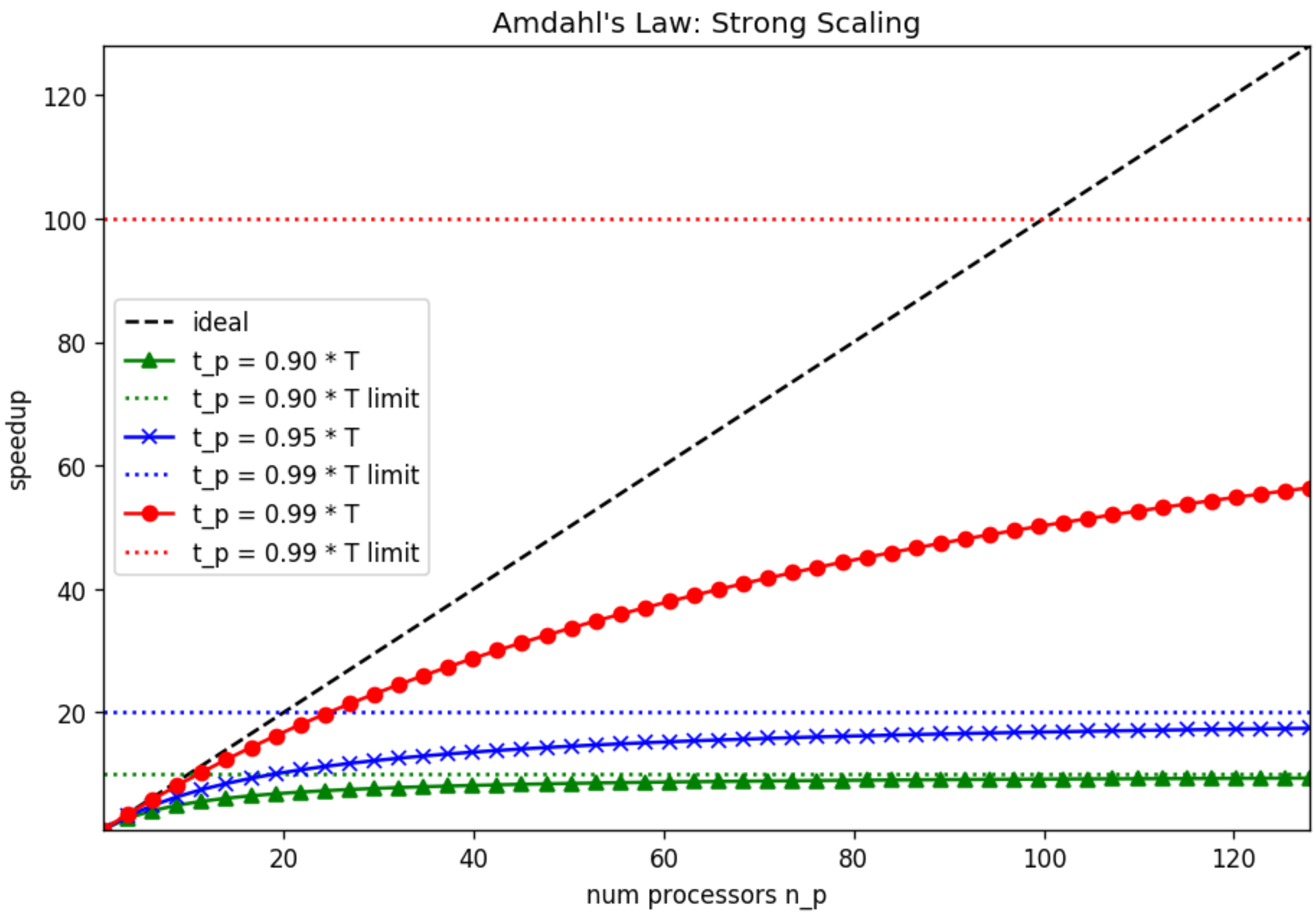
$$\eta_s(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$$

Performance Analysis: Scaling



In [16]:

```
from helpers import plot_amdahl_scaling
plot_amdahl_scaling( 1.0, 128.0 )
```



Performance Analysis: Conclusions from Amdahl's Law

- For a given problem with finite t_s, increases in performance are diminishing with rising n_p
- Controling and limiting t_s is crucial to achieve good parallel performance and speedup
- We assume that $t_s = const.$ for a given problem size. In practice, $t_s = f(n_p)$
- Even with t_s very small, scaling to 10^3 or even 10^4 of parallel processes (as in GPUs) under the assumptions of Amdahl is hard

Scaled Speedup, Gustafson-Barsis Law

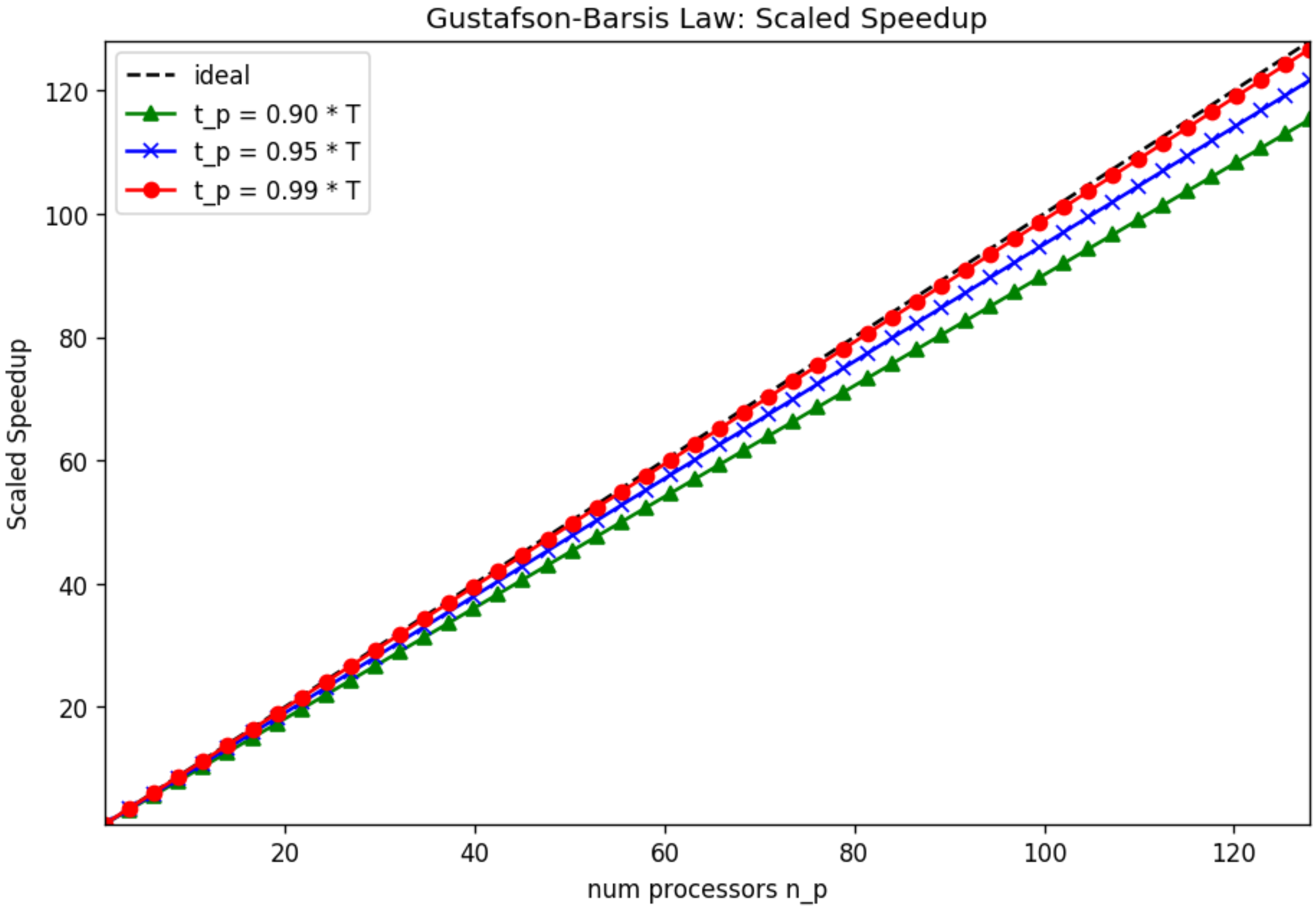
- Amdahl's Law is a bit pessimistic $\longleftarrow$ fixed problem size regardless of n_p
- What if we can **grow** the problem size together with rising number n_p?
- Scaled Speedup:

$$\eta(n_p) = f_s + n_p \cdot f_p$$

- $f_s = t_s/T$, fraction of the runtime that is serial
- $f_p = t_p/T$, fraction of the runtime that can be run in parallel on n_p processors.
- It holds that $f_s + f_p = 1$

In [18]:

```
from helpers import plot_gustafson_scaling
plot_gustafson_scaling( 1.0, 128.0 )
```

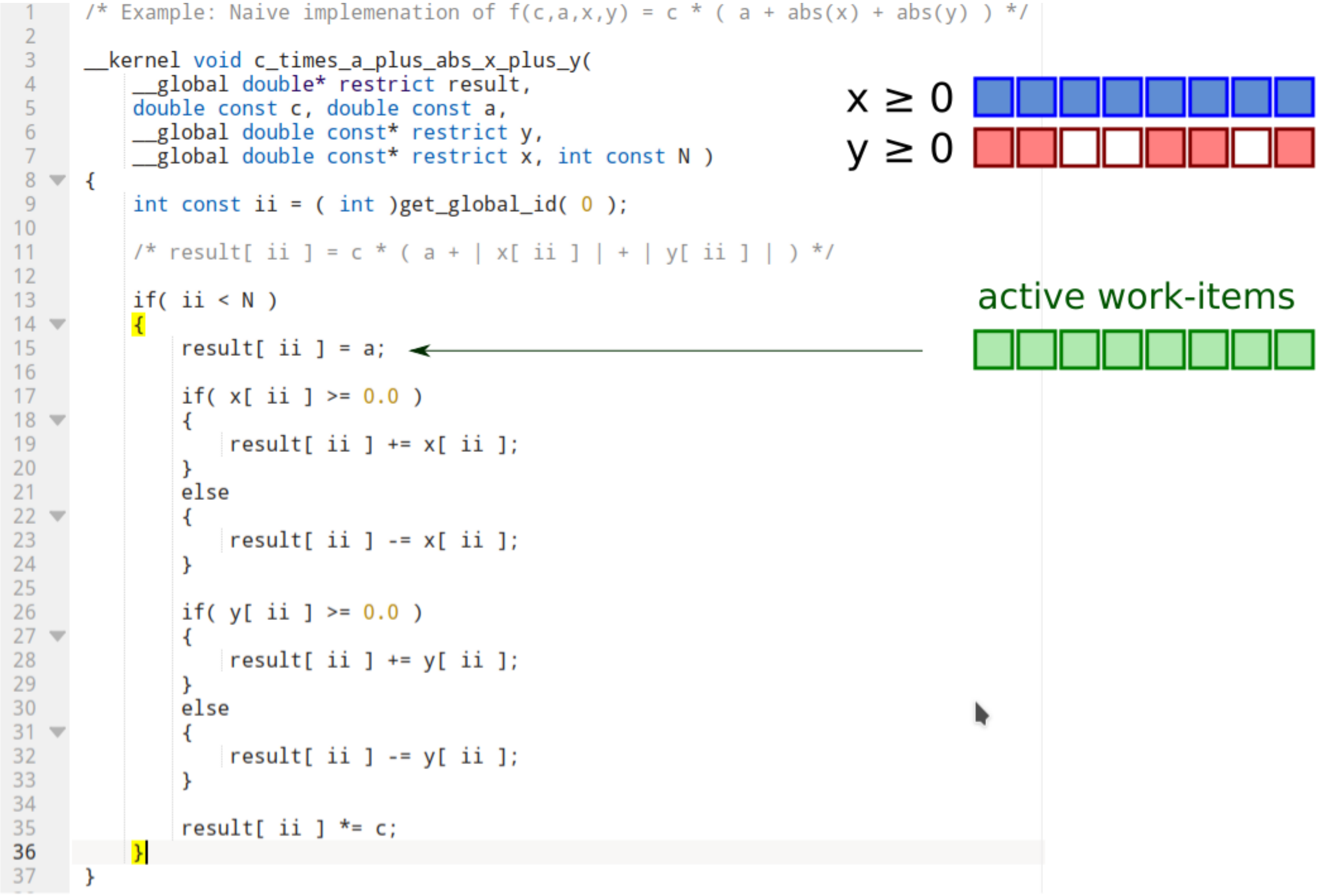


Contributions to t_s on GPUs

- Synchronization overhead between "processes" (threads) $\longrightarrow$ Problems that do not require communication between processes are more suitable to be parallelized (ideal: "Embarrisingly Parallel Problem")
- Latencies and transfer times for copying memory between host and device
- latencies in launching kernels
- Latencies for accessing resources (different memory regions!)
- During the runtime of the kernel: **thread divergence**

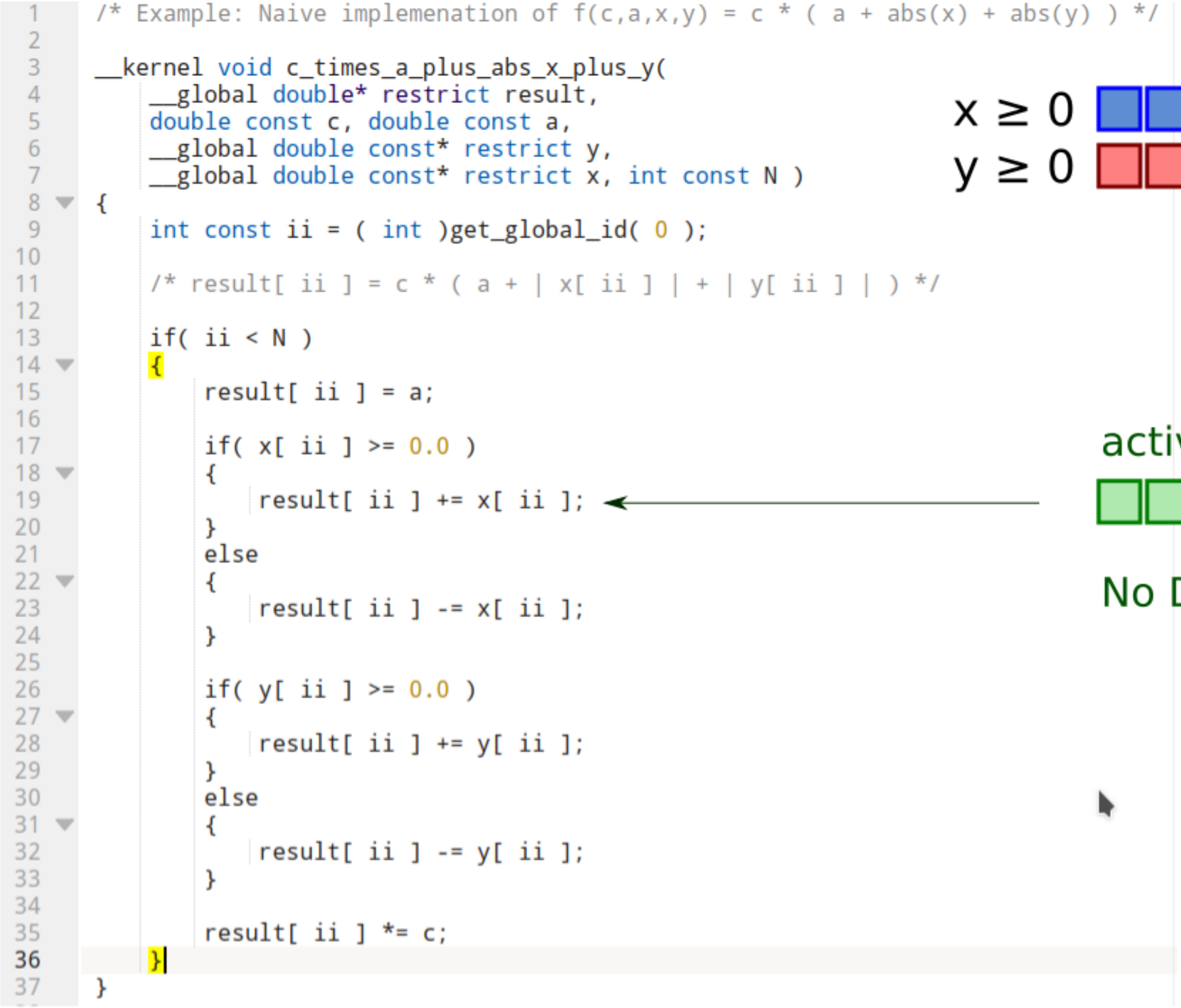
- Thread divergence is a consequence of the SPMD model typically implemented on GPUs
- Remember: Threads/work-items are organized in warps/wavefronts
- AMD: 64 work-items / wavefront
- NVIDIA: 32 threads/workitems / warp
- The GPU always executes a multiple of these numbers, even if you only ask to compute 1 work-items / threads
- All threads execute the **same program** in lock-step
- Dataparallelism: If the program branches depending on the data, code-paths have to be handled sequentially, thus increasing the t_s!

Example: Thread Divergence And Branching

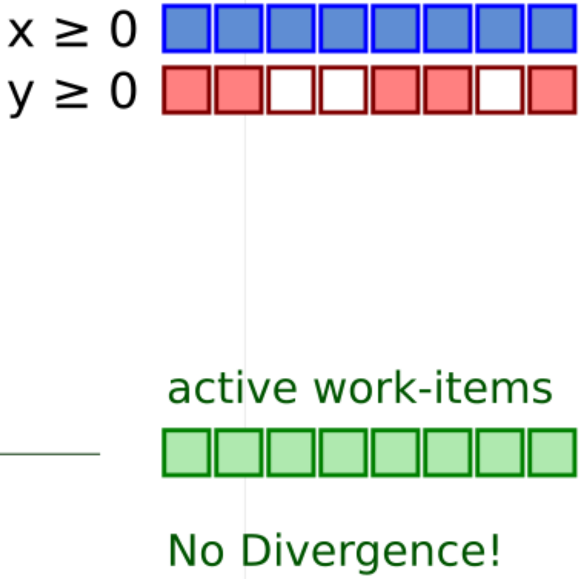


```
/* Example: Naive implemenation of f(c,a,x,y) = c * ( a + abs(x) + abs(y) ) */

__kernel void c_times_a_plus_abs_x_plus_y(
    __global double* restrict result,
    double const c, double const a,
    __global double const* restrict y,
    __global double const* restrict x, int const N )
{
    int const ii = ( int )get_global_id( 0 );

    /* result[ ii ] = c * ( a + | x[ ii ] | + | y[ ii ] | ) */

    if( ii < N )
    {
        result[ ii ] = a;

        if( x[ ii ] >= 0.0 )
        {
            result[ ii ] += x[ ii ];
        }
        else
        {
            result[ ii ] -= x[ ii ];
        }

        if( y[ ii ] >= 0.0 )
        {
            result[ ii ] += y[ ii ];
        }
        else
        {
            result[ ii ] -= y[ ii ];
        }

        result[ ii ] *= c;
    }
}
```

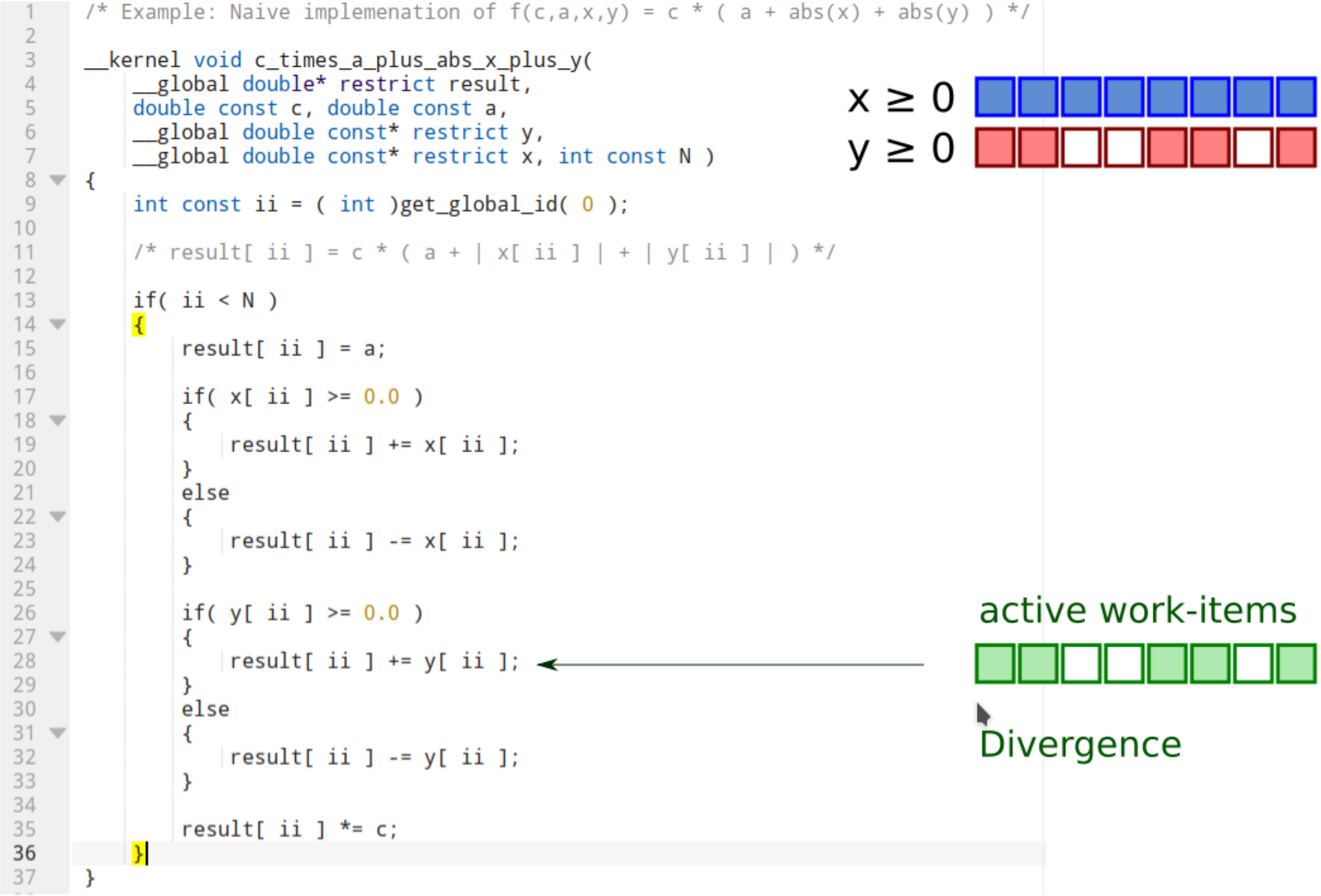
Example: Thread Divergence And Branching



```
/* Example: Naive implemenation of f(c,a,x,y) = c * ( a + abs(x) + abs(y) ) */

__kernel void c_times_a_plus_abs_x_plus_y(
    __global double* restrict result,
    double const c, double const a,
    __global double const* restrict y,
    __global double const* restrict x, int const N )
{
    int const ii = ( int )get_global_id( 0 );

    /* result[ ii ] = c * ( a + | x[ ii ] | + | y[ ii ] | ) */

    if( ii < N )
    {
        result[ ii ] = a;

        if( x[ ii ] >= 0.0 )
        {
            result[ ii ] += x[ ii ];
        }
        else
        {
            result[ ii ] -= x[ ii ];
        }

        if( y[ ii ] >= 0.0 )
        {
            result[ ii ] += y[ ii ];
        }
        else
        {
            result[ ii ] -= y[ ii ];
        }

        result[ ii ] *= c;
    }
}
```

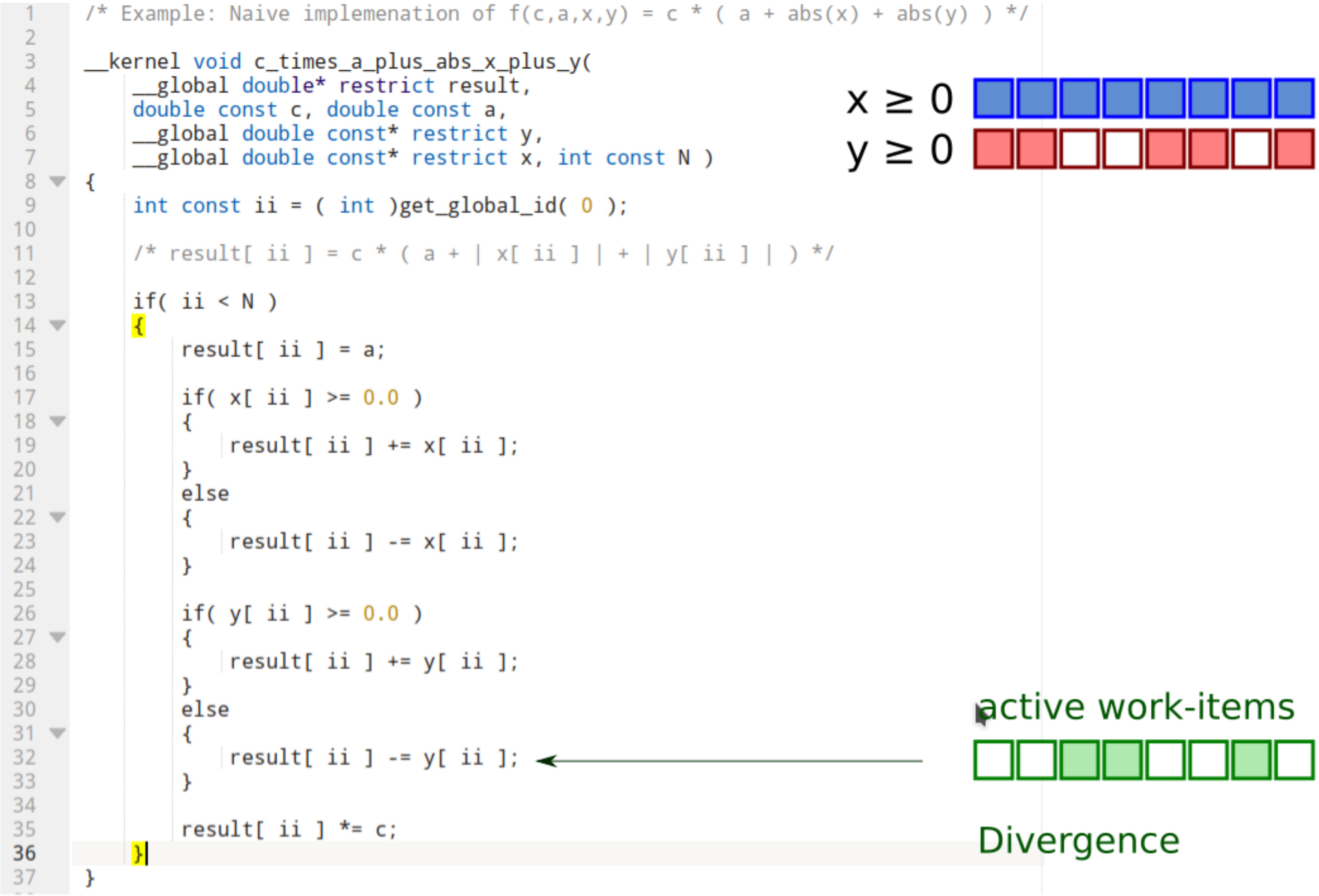


Example: Thread Divergence And Branching



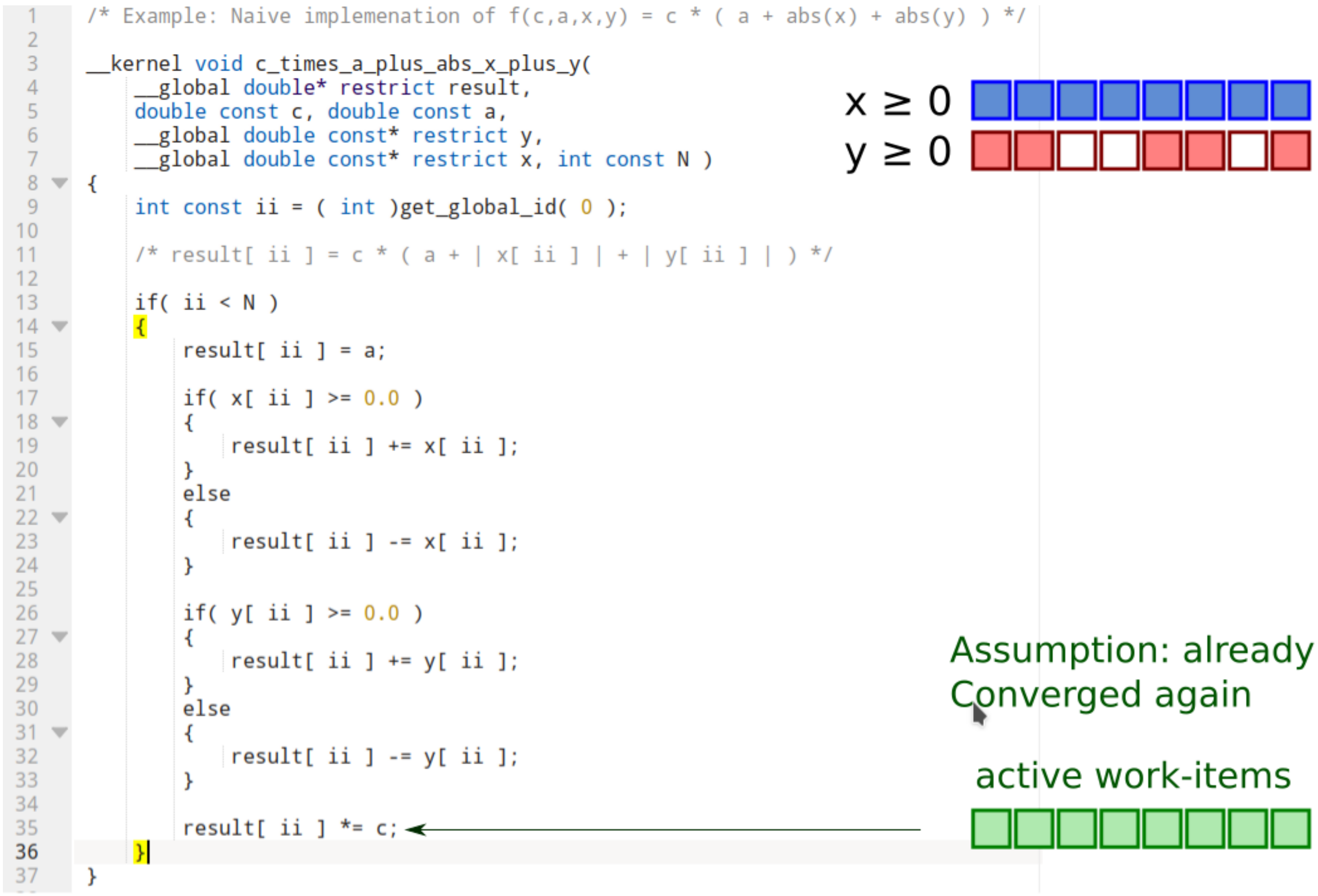
```
/* Example: Naive implemenation of f(c,a,x,y) = c * ( a + abs(x) + abs(y) ) */

__kernel void c_times_a_plus_abs_x_plus_y(
    __global double* restrict result,
    double const c, double const a,
    __global double const* restrict y,
    __global double const* restrict x, int const N )
{
    int const ii = ( int )get_global_id( 0 );

    /* result[ ii ] = c * ( a + | x[ ii ] | + | y[ ii ] | ) */

    if( ii < N )
    {
        result[ ii ] = a;

        if( x[ ii ] >= 0.0 )
        {
            result[ ii ] += x[ ii ];
        }
        else
        {
            result[ ii ] -= x[ ii ];
        }

        if( y[ ii ] >= 0.0 )
        {
            result[ ii ] += y[ ii ];
        }
        else
        {
            result[ ii ] -= y[ ii ];
        }

        result[ ii ] *= c;
    }
}
```

Example: Thread Divergence And Branching



```
/* Example: Naive implemenation of f(c,a,x,y) = c * ( a + abs(x) + abs(y) ) */

__kernel void c_times_a_plus_abs_x_plus_y(
    __global double* restrict result,
    double const c, double const a,
    __global double const* restrict y,
    __global double const* restrict x, int const N )
{
    int const ii = ( int )get_global_id( 0 );

    /* result[ ii ] = c * ( a + | x[ ii ] | + | y[ ii ] | ) */

    if( ii < N )
    {
        result[ ii ] = a;

        if( x[ ii ] >= 0.0 )
        {
            result[ ii ] += x[ ii ];
        }
        else
        {
            result[ ii ] -= x[ ii ];
        }

        if( y[ ii ] >= 0.0 )
        {
            result[ ii ] += y[ ii ];
        }
        else
        {
            result[ ii ] -= y[ ii ];
        }

        result[ ii ] *= c;
    }
}
```

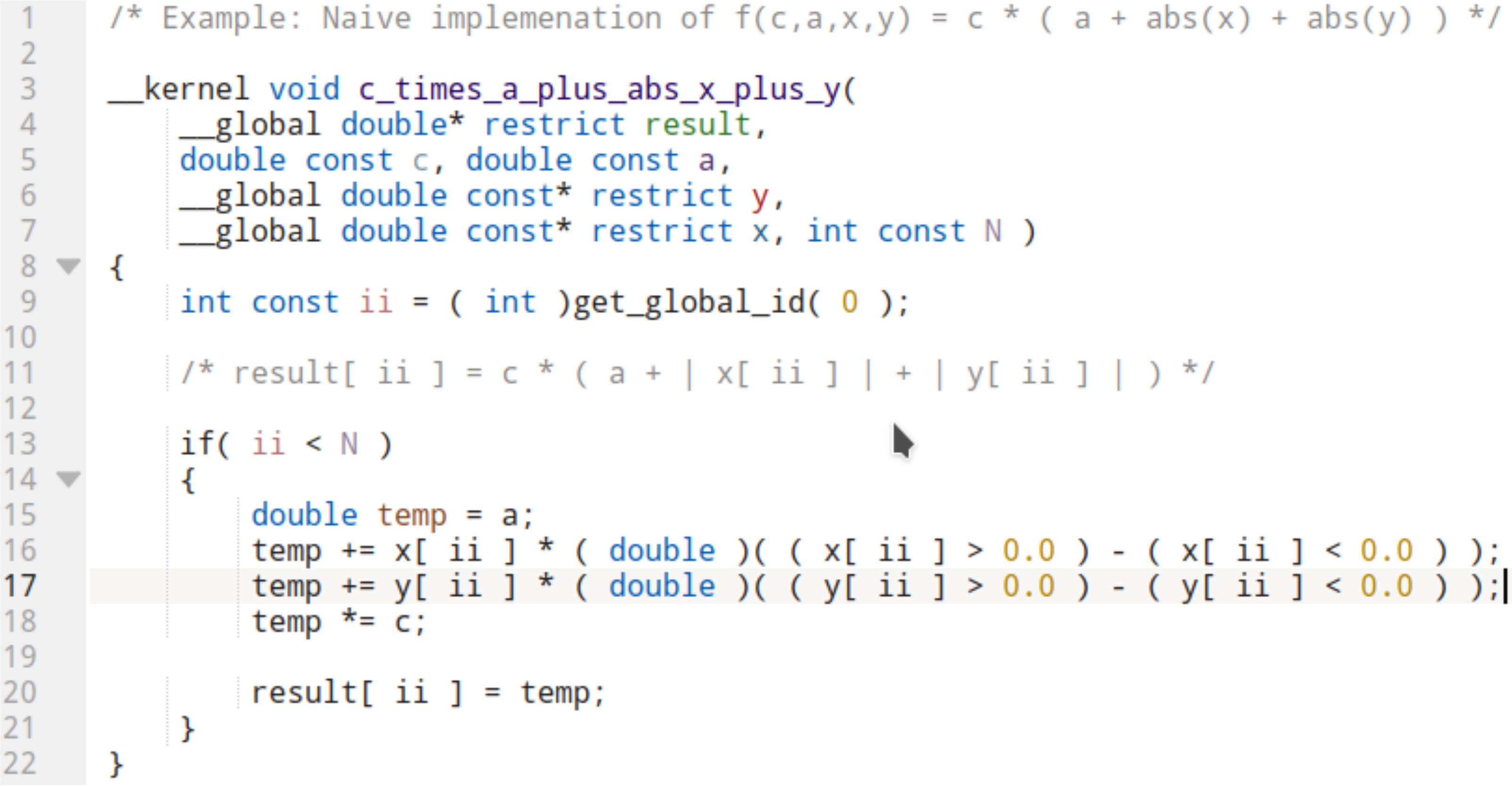
Example: Thread Divergence And Branching



```
/* Example: Naive implemenation of f(c,a,x,y) = c * ( a + abs(x) + abs(y) ) */

__kernel void c_times_a_plus_abs_x_plus_y(
    __global double* restrict result,
    double const c, double const a,
    __global double const* restrict y,
    __global double const* restrict x, int const N )
{
    int const ii = ( int )get_global_id( 0 );

    /* result[ ii ] = c * ( a + | x[ ii ] | + | y[ ii ] | ) */

    if( ii < N )
    {
        result[ ii ] = a;

        if( x[ ii ] >= 0.0 )
        {
            result[ ii ] += x[ ii ];
        }
        else
        {
            result[ ii ] -= x[ ii ];
        }

        if( y[ ii ] >= 0.0 )
        {
            result[ ii ] += y[ ii ];
        }
        else
        {
            result[ ii ] -= y[ ii ];
        }

        result[ ii ] *= c;
    }
}
```

Assumption: already Converged again

Example: Alternative, Branchless Formulation of Kernel



```
/* Example: Naive implemenation of f(c,a,x,y) = c * ( a + abs(x) + abs(y) ) */

__kernel void c_times_a_plus_abs_x_plus_y(
    __global double* restrict result,
    double const c, double const a,
    __global double const* restrict y,
    __global double const* restrict x, int const N )
{
    int const ii = ( int )get_global_id( 0 );

    /* result[ ii ] = c * ( a + | x[ ii ] | + | y[ ii ] | ) */

    if( ii < N )
    {
        double temp = a;
        temp += x[ ii ] * ( double )( ( x[ ii ] > 0.0 ) - ( x[ ii ] < 0.0 ) );
        temp += y[ ii ] * ( double )( ( y[ ii ] > 0.0 ) - ( y[ ii ] < 0.0 ) );
        temp *= c;

        result[ ii ] = temp;
    }
}
```

Summary & Conclusions

- GPUs offer a large number of threads and significant floating-point computing power
- There are many options to get started with GPU programming without starting to go fully towards low-level programming
- The two current main frameworks, OpenCL and CUDA, differ in scope and implementation but are similar enough that a common implementation of Kernels is possible
- Controlling and Limiting the sequential run-time component is crucial to achieve good scaling and parallel performance
- Thread Divergence can be a key contributor to the sequential runtime-portion
- Not covered: optimal grid dimensions and hardware utilization

Thank You For Your Attention!