GPU Computing Tutorial

M. Schwinzerl, R. De Maria

CERN

riccardo.de.maria@cern.ch, martin.schwinzerl@cern.ch

December 5, 2018

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Outline

- Goal
  - Identify Problems benefitting from GPU Computing
  - Understand the typical structure of a GPU enhanced Program
  - Introduce the two most established Frameworks
  - Basic Idea about Performance Analysis
  - Teaser: SixTrackLib
  - Code: `https://github.com/martinschwinzerl/intro_gpu_computing`

Introduction: CPUs vs. GPUs (Spoilers Ahead!)

GPUs have:

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

Introduction: CPUs vs. GPUs (Spoilers Ahead!)

GPUs have:

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

Introduction: CPUs vs. GPUs (Spoilers Ahead!)

GPUs have:

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

Introduction: CPUs vs. GPUs (Spoilers Ahead!)

GPUs have:

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

Introduction: CPUs vs. GPUs (Spoilers Ahead!)

GPUs have:

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

GPU hardware types:

- Low-end Gaming: <100$, still equal or better computing than desktop CPU in SP and DP
- High-end Gaming : 100-800$, substantial computing in SP, DP= 1/32 SP
- Server HPC: 6000-8000$, substantial computing in SP, DP= 1/2 SP, no video
- Hybrid HPC: 3000-8000$, substantial computing in SP, DP= 1/2 SP
- Server AI: 6000-8000$, substantial computing in SP, DP= 1/32 SP, no video

Introduction: CPUs vs. GPUs (Spoilers Ahead!)

GPUs have:

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

GPU hardware types:

- Low-end Gaming: <100$, still equal or better computing than desktop CPU in SP and DP
- High-end Gaming : 100-800$, substantial computing in SP, DP= 1/32 SP
- Server HPC: 6000-8000$, substantial computing in SP, DP= 1/2 SP, no video
- Hybrid HPC: 3000-8000$, substantial computing in SP, DP= 1/2 SP
- Server AI: 6000-8000$, substantial computing in SP, DP= 1/32 SP, no video

Introduction: GPU vs CPU (Example)

Specification	CPU	GPU
Cores or compute units	2-64	16-80
Arithmetic units per core	2-8	64
GFlops SP	24-2000	1000 - 15000
GFlops DP	12-1000	100 - 7500

- Obtaining theoretical performance is guaranteed only for a limited number of problems
- Code complexity, Number of temporary varaibles, Branches, Latencies, Memory access patterns / Synchronization, ...

- Question 1: How to write and run programs for GPUs?
- Question 2: What kind of problems are suitable for GPU computing?

Introduction: GPU vs CPU (Example)

Specification	CPU	GPU
Cores or compute units	2-64	16-80
Arithmetic units per core	2-8	64
GFlops SP	24-2000	1000 - 15000
GFlops DP	12-1000	100 - 7500

- Obtaining theoretical performance is guaranteed only for a limited number of problems
- Code complexity, Number of temporary varaibles, Branches, Latencies, Memory access patterns / Synchronization, ...
- Question 1: How to write and run programs for GPUs?
- Question 2: What kind of problems are suitable for GPU computing?

Introduction: GPU vs CPU (Example)

Specification	CPU	GPU
Cores or compute units	2-64	16-80
Arithmetic units per core	2-8	64
GFlops SP	24-2000	1000 - 15000
GFlops DP	12-1000	100 - 7500

- Obtaining theoretical performance is guaranteed only for a limited number of problems
- Code complexity, Number of temporary varaibles, Branches, Latencies, Memory access patterns / Synchronization, ...

- Question 1: How to write and run programs for GPUs?
- Question 2: What kind of problems are suitable for GPU computing?

Introduction: GPU vs CPU (Example)

Specification	CPU	GPU
Cores or compute units	2-64	16-80
Arithmetic units per core	2-8	64
GFlops SP	24-2000	1000 - 15000
GFlops DP	12-1000	100 - 7500

- Obtaining theoretical performance is guaranteed only for a limited number of problems
- Code complexity, Number of temporary varaibles, Branches, Latencies, Memory access patterns / Synchronization, ...
- Question 1: How to write and run programs for GPUs?
- Question 2: What kind of problems are suitable for GPU computing?

Introduction: GPU vs CPU (Example)

Specification	CPU	GPU
Cores or compute units	2-64	16-80
Arithmetic units per core	2-8	64
GFlops SP	24-2000	1000 - 15000
GFlops DP	12-1000	100 - 7500

- Obtaining theoretical performance is guaranteed only for a limited number of problems
- Code complexity, Number of temporary varaibles, Branches, Latencies, Memory access patterns / Synchronization, ...

- Question 1: How to write and run programs for GPUs?
- Question 2: What kind of problems are suitable for GPU computing?

Non-Scary GPU programming

CuPy: Cuda + Python https://cupy.chainer.org/

```
import cupy as cp
x = cp.arange(6).reshape(2, 3).astype('f')
print( x )
% array([[ 0.,  1.,  2.],
%        [ 3.,  4.,  5.]], dtype=float32)

x.sum(axis=1)
% array([  3.,  12.], dtype=float32)
```

Overview: Coding frameworks

- Cuda: NVIDIA, GPUs only, C99 / C++1x language, unified memory (free)
- OpenCL: vendor and hardware neutral (AMD, NVIDIA, Intel, CPU(pocl, Intel)), v1.2: C99-like language (free)
- SyCL: open standard, beta commercial implementation (Intel, AMD) (no NVIDIA)
- OpenACC and OpenMP: directive based, offloading, open standard, compiler directives
- ROCm: open source (from AMD) (HIP compatible AMD, Nvidia)
- clang+SPIRV+vulkan: vendor neutral (AMD, NVIDIA) not easy
- clang+PTX: Nvidia not easy
- numba, cupy (pure python)
- pyopencl (python + OpenCL kernels), pycuda (python + Cuda kernels)

Overview: Coding frameworks

- Cuda: NVIDIA, GPUs only, C99 / C++1x language, unified memory (free)
- OpenCL: vendor and hardware neutral (AMD, NVIDIA, Intel, CPU(pocl, Intel)), v1.2: C99-like language (free)
- SyCL: open standard, beta commercial implementation (Intel, AMD) (no NVIDIA)
- OpenACC and OpenMP: directive based, offloading, open standard, compiler directives
- ROCm: open source (from AMD) (HIP compatible AMD, Nvidia)
- clang+SPIRV+vulkan: vendor neutral (AMD, NVIDIA) not easy
- clang+PTX: Nvidia not easy
- numba, cupy (pure python)
- pyopencl (python + OpenCL kernels), pycuda (python + Cuda kernels)

Overview: Coding frameworks

- Cuda: NVIDIA, GPUs only, C99 / C++1x language, unified memory (free)
- OpenCL: vendor and hardware neutral (AMD, NVIDIA, Intel, CPU(pocl, Intel)), v1.2: C99-like language (free)
- SyCL: open standard, beta commercial implementation (Intel, AMD) (no NVIDIA)
- OpenACC and OpenMP: directive based, offloading, open standard, compiler directives
- ROCm: open source (from AMD) (HIP compatible AMD, Nvidia)
- clang+SPIRV+vulkan: vendor neutral (AMD, NVIDIA) not easy
- clang+PTX: Nvidia not easy
- numba, cupy (pure python)
- pyopencl (python + OpenCL kernels), pycuda (python + Cuda kernels)

Overview: Coding frameworks

- Cuda: NVIDIA, GPUs only, C99 / C++1x language, unified memory (free)
- OpenCL: vendor and hardware neutral (AMD, NVIDIA, Intel, CPU(pocl, Intel)), v1.2: C99-like language (free)
- SyCL: open standard, beta commercial implementation (Intel, AMD) (no NVIDIA)
- OpenACC and OpenMP: directive based, offloading, open standard, compiler directives
- ROCm: open source (from AMD) (HIP compatible AMD, Nvidia)
- clang+SPIRV+vulkan: vendor neutral (AMD, NVIDIA) not easy
- clang+PTX: Nvidia not easy
- numba, cupy (pure python)
- pyopencl (python + OpenCL kernels), pycuda (python + Cuda kernels)

Overview: Coding frameworks

- Cuda: NVIDIA, GPUs only, C99 / C++1x language, unified memory (free)
- OpenCL: vendor and hardware neutral (AMD, NVIDIA, Intel, CPU(pocl, Intel)), v1.2: C99-like language (free)
- SyCL: open standard, beta commercial implementation (Intel, AMD) (no NVIDIA)
- OpenACC and OpenMP: directive based, offloading, open standard, compiler directives
- ROCm: open source (from AMD) (HIP compatible AMD, Nvidia)
- clang+SPIRV+vulkan: vendor neutral (AMD, NVIDIA) not easy
- clang+PTX: Nvidia not easy
- numba, cupy (pure python)
- pyopencl (python + OpenCL kernels), pycuda (python + Cuda kernels)

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

OpenCL

- Hardware Neutral: GPUs, CPUs, FPGAs, Signal Processors
- Standardized by Khronos Group
- Programs should work across implementations $\rightarrow$ "Least Common Denominator' Programming"
- Now: OpenCL 1.2 (C99 - like language for device programs), future: OpenCL 2.x, OpenCL-Next?
- Run-time compilation

Cuda

- Hardware: only NVIDIA GPUs
- Software: only implementation from NVIDIA,
- Tools, Libraries and Toolkits also available
- Software and hardware tightly integrated (versioning!)
- No emulator/CPU target on recent implementations
- Version 10.0 with C99 and C++11/14 like language
- Single-file compilation

Comparison: OpenCL and Cuda

opencl/vec_add_opencl.cpp

```cpp
#include <algorithm>
/* .... */
#include <vector>

#include <CL/cl2.hpp>

int main( void )
{
    /* ---------------------------------------------------------------------- */
    /* prepare the host vectors: */

    int32_t const N = int32_t{ 10000 };
    std::vector< double > x( N, double{ 0.0 } );
    std::vector< double > y( N, double{ 0.0 } );
    std::vector< double > z( N, double{ 0.0 } );

    std::mt19937_64 prng( 20181205u );
    std::uniform_real_distribution< double > dist( -10., +10. );

    for( int32_t ii = int32_t{ 0 } ; ii < N ; ++ii )
    {
        x[ ii ] = dist( prng ); y[ ii ] = dist( prng );
    }

    /* ---------------------------------------------------------------------- */
    /* Select the first device on the first platform: */
    std::vector< cl::Platform > platforms;
    cl::Platform::get( &platforms );
    std::vector< cl::Device > devices;

    bool device_found = false;
    cl::Device device;

    for( auto const& available_platform : platforms )
    {
        devices.clear();
        available_platform.getDevices( CL_DEVICE_TYPE_ALL, &devices );

        if( !devices.empty() )
        {
            device = devices.front();
            device_found = true;
            break;
        }
    }
    assert( device_found );

    /* ---------------------------------------------------------------------- */
    /* Build the program and get the kernel: */

    cl::Context context( device );
    cl::CommandQueue queue( context, device );

    std::string const kernel_source =
    "__kernel void add_vec_kernel( \r\n"
        " __global double const* restrict x, __global double const* restrict y,"
        " __global double* restrict z, int const n )\r\n"
    "{\r\n"
    "   int const gid = get_global_id( 0 );\r\n"
    "   if( gid < n ) {\r\n"
    "       z[ gid ] = x[ gid ] + y[ gid ]; \r\n"
    "   }\r\n"
    "}\r\n";

    std::string const compile_options = "-w -Werror";

    cl::Program program( context, kernel_source );
    cl_int ret = program.build( compile_options.c_str() );

    assert( ret == CL_SUCCESS );
    cl::Kernel kernel( program, "add_vec_kernel" );

    /* ---------------------------------------------------------------------- */
    /* Allocate the buffers on the device */
    /* x_arg, y_arg, z_arg ... handles on the host side managing buffers in *
     * the device memory */

    cl::Buffer x_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
    cl::Buffer y_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
    cl::Buffer z_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );

    /* Transfer x and y from the host to the device */

    ret = queue.enqueueWriteBuffer( x_arg, CL_TRUE, std::size_t{ 0 },
                                    x.size() * sizeof( double ), x.data() );
    assert( ret == CL_SUCCESS );

    ret = queue.enqueueWriteBuffer( y_arg, CL_TRUE, std::size_t{ 0 },
                                    y.size() * sizeof( double ), y.data() );
    assert( ret == CL_SUCCESS );

    /* ---------------------------------------------------------------------- */
    /* Prepare the kernel for execution: bind the arguments to the kernel */

    kernel.setArg( 0, x_arg ); kernel.setArg( 1, y_arg );
    kernel.setArg( 2, z_arg ); kernel.setArg( 3, N );

    /* ---------------------------------------------------------------------- */
    /* execute the kernel on the device */
    cl::NDRange offset = cl::NullRange; cl::NDRange local  = cl::NullRange;

    ret = queue.enqueueNDRangeKernel( kernel, offset, N, local );
    assert( ret == CL_SUCCESS );

    /* ---------------------------------------------------------------------- */
    /* transfer the result from the device buffer to the host buffer */

    ret = queue.enqueueReadBuffer( z_arg, CL_TRUE, std::size_t{ 0 },
                                   z.size() * sizeof( double ), z.data() );

    /* ---------------------------------------------------------------------- */
    /* verify that the result is correct */

    bool success = true;
    double const EPS = std::numeric_limits< double >::epsilon();

    for( int32_t ii = int32_t{ 0 } ; ii < N ; ++ii )
    {
        if( std::fabs( ( x[ ii ] + y[ ii ] ) - z[ ii ] ) > EPS )
        {
            success = false;
            break;
        }
    }

    std::cout << "Success: " << std::boolalpha << success << std::endl;
    return 0;
}
```

Comparison: OpenCL and Cuda

opencl/vec_add_opencl.cpp

```cpp
#include <algorithm>
/* .... */
#include <vector>

#include <CL/cl2.hpp>

int main( void )
{
    /* ---------------------------------------------------------------------- */
    /* prepare the host vectors: */

    int32_t const N = int32_t{ 10000 };
    std::vector< double > x( N, double{ 0.0 } );
    std::vector< double > y( N, double{ 0.0 } );
    std::vector< double > z( N, double{ 0.0 } );

    std::mt19937_64 prng( 20181205u );
    std::uniform_real_distribution< double > dist( -10., +10. );

    for( int32_t ii = int32_t{ 0 } ; ii < N ; ++ii )
    {
        x[ ii ] = dist( prng ); y[ ii ] = dist( prng );
    }

    /* ---------------------------------------------------------------------- */
    /* Select the first device on the first platform: */
    std::vector< cl::Platform > platforms;
    cl::Platform::get( &platforms );
    std::vector< cl::Device > devices;

    bool device_found = false;
    cl::Device device;

    for( auto const& available_platform : platforms )
    {
        devices.clear();
        available_platform.getDevices( CL_DEVICE_TYPE_ALL, &devices );

        if( !devices.empty() )
        {
            device = devices.front();
            device_found = true;
            break;
        }
    }
    assert( device_found );

    /* ---------------------------------------------------------------------- */
    /* Build the program and get the kernel: */

    cl::Context context( device );
    cl::CommandQueue queue( context, device );

    std::string const kernel_source =
    "__kernel void add_vec_kernel( \r\n"
        " __global double const* restrict x, __global double const* restrict y,"
        " __global double* restrict z, int const n )\r\n"
    "{\r\n"
    "    int const gid = get_global_id( 0 );\r\n"
    "    if( gid < n ) {\r\n"
    "        z[ gid ] = x[ gid ] + y[ gid ]; \r\n"
    "    }\r\n"
    "}\r\n";

    std::string const compile_options = "-w -Werror";

    cl::Program program( context, kernel_source );
    cl_int ret = program.build( compile_options.c_str() );

    assert( ret == CL_SUCCESS );
    cl::Kernel kernel( program, "add_vec_kernel" );

    /* ---------------------------------------------------------------------- */
    /* Allocate the buffers on the device */
    /* x_arg, y_arg, z_arg ... handles on the host side managing buffers in *
     * the device memory */

    cl::Buffer x_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
    cl::Buffer y_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
    cl::Buffer z_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );

    /* Transfer x and y from the host to the device */

    ret = queue.enqueueWriteBuffer( x_arg, CL_TRUE, std::size_t{ 0 },
                                    x.size() * sizeof( double ), x.data() );
    assert( ret == CL_SUCCESS );

    ret = queue.enqueueWriteBuffer( y_arg, CL_TRUE, std::size_t{ 0 },
                                    y.size() * sizeof( double ), y.data() );
    assert( ret == CL_SUCCESS );

    /* ---------------------------------------------------------------------- */
    /* Prepare the kernel for execution: bind the arguments to the kernel */

    kernel.setArg( 0, x_arg ); kernel.setArg( 1, y_arg );
    kernel.setArg( 2, z_arg ); kernel.setArg( 3, N );

    /* ---------------------------------------------------------------------- */
    /* execute the kernel on the device */
    cl::NDRange offset = cl::NullRange; cl::NDRange local  = cl::NullRange;

    ret = queue.enqueueNDRangeKernel( kernel, offset, N, local );
    assert( ret == CL_SUCCESS );

    /* ---------------------------------------------------------------------- */
    /* transfer the result from the device buffer to the host buffer */

    ret = queue.enqueueReadBuffer( z_arg, CL_TRUE, std::size_t{ 0 },
                                   z.size() * sizeof( double ), z.data() );

    /* ---------------------------------------------------------------------- */
    /* verify that the result is correct */

    bool success = true;
    double const EPS = std::numeric_limits< double >::epsilon();

    for( int32_t ii = int32_t{ 0 } ; ii < N ; ++ii )
    {
        if( std::fabs( ( x[ ii ] + y[ ii ] ) - z[ ii ] ) > EPS )
        {
            success = false;
            break;
        }
    }

    std::cout << "Success: " << std::boolalpha << success << std::endl;
    return 0;
}
```

Comparison: OpenCL and Cuda

cuda/vec_add_cuda.cu

```cpp
/* ..... */
#include <vector>

#include <cuda_runtime_api.h>
#include <cuda.h>

/* Define the kernel function: */

__global__ void add_vec_kernel(
    double const* __restrict__ x, double const* __restrict__ y,
    double* __restrict__ z, int const n )
{
    /* blockIdx, blockDim and threadIdx are variables describing the
     * dimensions of the "grid" which are automatically provided by the
     * Cuda runtime */

    int const gid = blockIdx.x * blockDim.x + threadIdx.x;

    if( gid < n )
    {
        z[ gid ] = x[ gid ] + y[ gid ];
    }

    return;
}

int main( void )
{
    /* ------------------------------------------------------------------ */
    /* prepare the host vectors: */

    int32_t const N = int32_t{ 10000 };

    std::vector< double > x( N, double{ 0.0 } );
    std::vector< double > y( N, double{ 0.0 } );
    std::vector< double > z( N, double{ 0.0 } );

    std::mt19937_64 prng( 20181205u );
    std::uniform_real_distribution< double >
        dist( double{ -10. }, double{ +10. } );

    for( int32_t ii = int32_t{ 0 } ; ii < N ; ++ii )
    {
        x[ ii ] = dist( prng );
        y[ ii ] = dist( prng );
    }

    cudaError_t cu_err;

    /* ------------------------------------------------------------------ */
    /* use the "default" / "first" Cuda device for the program: */

    int device = int{ 0 };
    ::cudaGetDevice( &device );
    cu_err = ::cudaDeviceSynchronize();
    assert( cu_err == ::cudaSuccess );

    /* ------------------------------------------------------------------ */
    /* Allocate the buffers on the device */
    /* x_arg, y_arg, z_arg ... handles on the host side managing buffers in *
     * the device memory */

    double* x_arg = nullptr;
    double* y_arg = nullptr;
    double* z_arg = nullptr;

    ::cudaMalloc( &x_arg, sizeof( double ) * N );
    ::cudaMalloc( &y_arg, sizeof( double ) * N );
    ::cudaMalloc( &z_arg, sizeof( double ) * N );

    /* ------------------------------------------------------------------ */
    /* Transfer x and y from host to device */

    ::cudaMemcpy( x_arg, x.data(), sizeof( double ) * N, cudaMemcpyHostToDevice );
    ::cudaMemcpy( y_arg, y.data(), sizeof( double ) * N, cudaMemcpyHostToDevice );

    /* ------------------------------------------------------------------ */
    /* execute kernel on the device */

    int32_t const threads_per_block = int32_t{ 128 };

    int32_t const num_blocks =
        ( N + threads_per_block - int32_t{ 1 } ) / threads_per_block;

    add_vec_kernel<<< num_blocks, threads_per_block >>>( x_arg, y_arg, z_arg, N );

    cu_err = ::cudaPeekAtLastError();
    assert( cu_err == ::cudaSuccess );

    /* ------------------------------------------------------------------ */
    /* transfer the result from the device buffer to the host buffer */

    ::cudaMemcpy( z.data(), z_arg, sizeof( double ) * N, cudaMemcpyDeviceToHost );

    /* ------------------------------------------------------------------ */
    /* verify that the result is correct */

    bool success = true;
    double const EPS = std::numeric_limits< double >::epsilon();

    for( int32_t ii = int32_t{ 0 } ; ii < N ; ++ii )
    {
        if( std::fabs( ( x[ ii ] + y[ ii ] ) - z[ ii ] ) > EPS )
        {
            success = false;
            break;
        }
    }

    std::cout << "Success: " << std::boolalpha << success << std::endl;

    /* ------------------------------------------------------------------ */
    /* Clean-up */

    ::cudaFree( x_arg );
    ::cudaFree( y_arg );
    ::cudaFree( z_arg );

    return 0;
}
```

Comparison: OpenCL and Cuda

Comparison: OpenCL and Cuda

OpenCL

- **Primarily**: Run-time compilation
- Single-file compilation available (SyCL)

Cuda

- **Primarily**: Single-file compilation
- Run-time compilation available (NVRTC)

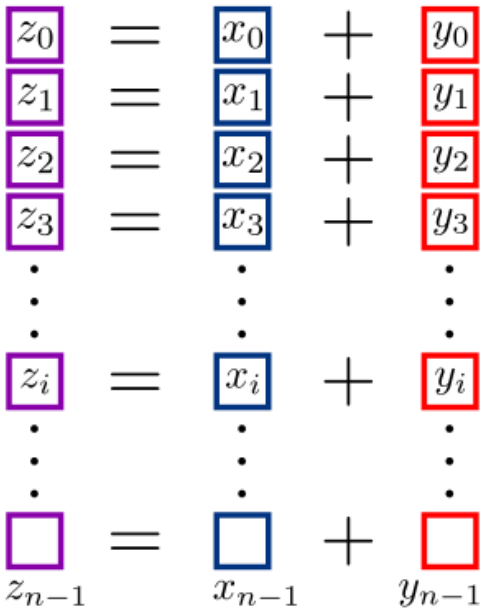
Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

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

Example: Vector-Vector Addition (Overview)

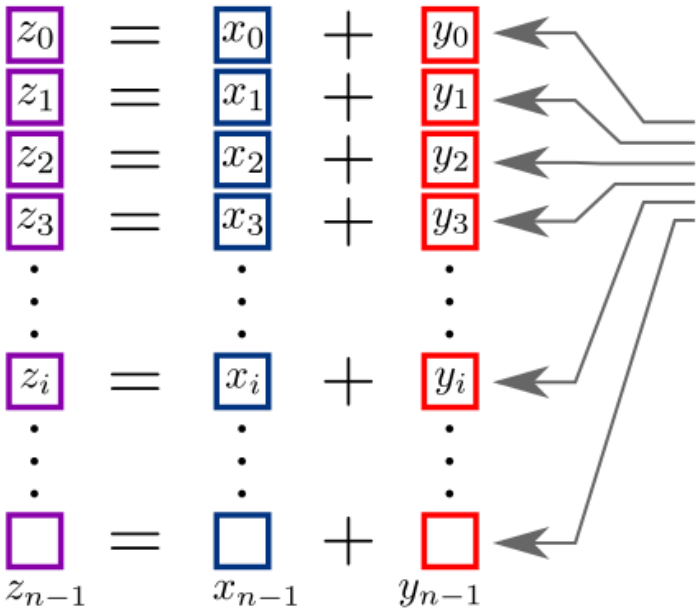
$$\vec{z} = \vec{x} + \vec{y}$$



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

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

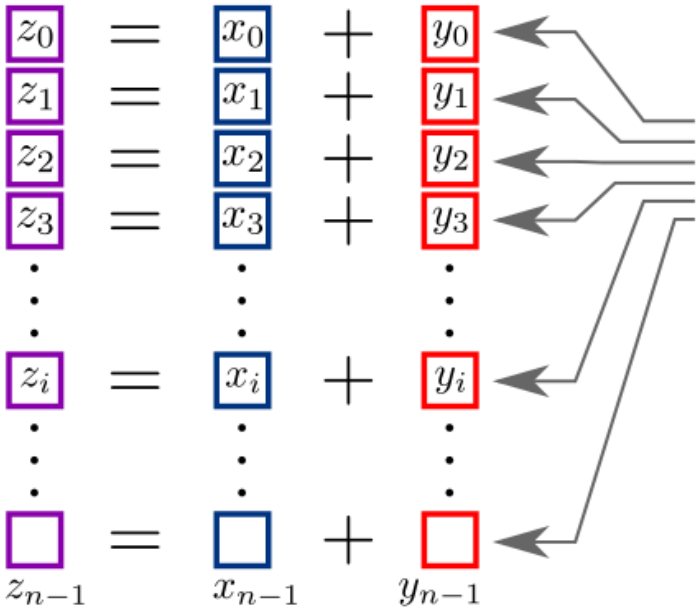


$\vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$

- Independent Operations
- Operations of comparable "cost"

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$



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

"Inherrent Parallelism"

"Embarrassingly Parallel"

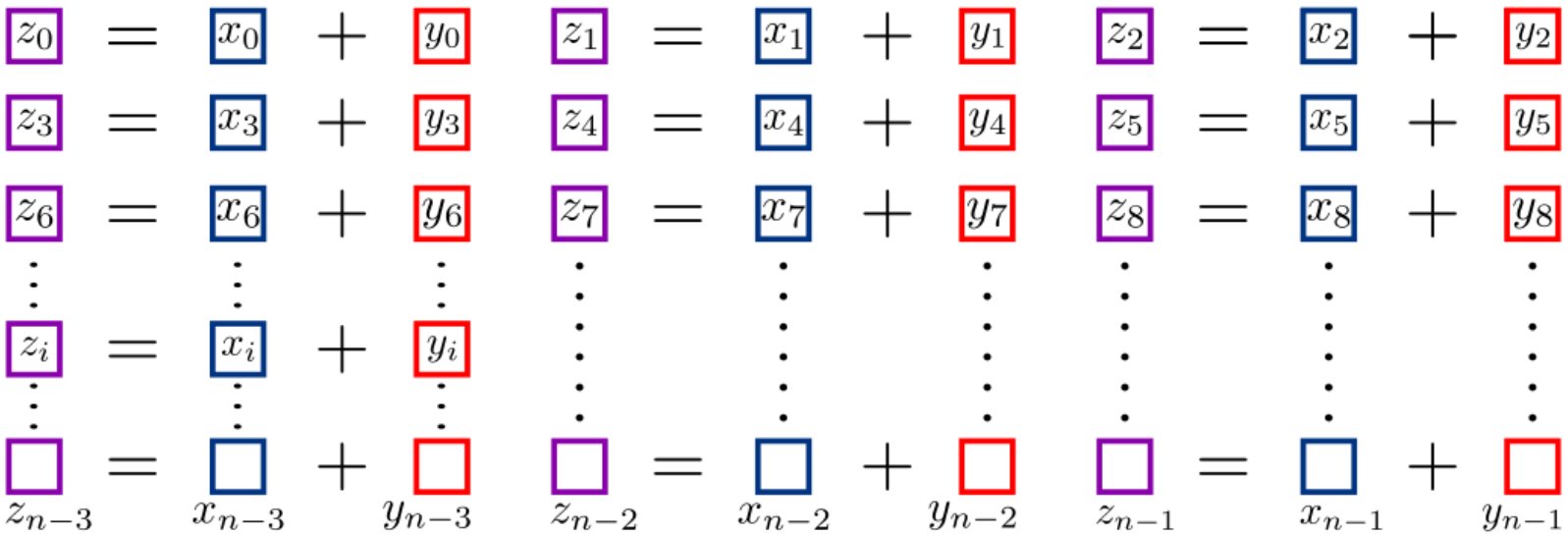
- Independent Operations
- Operations of comparable "cost"

Example: Vector-Vector Addition (Overview)

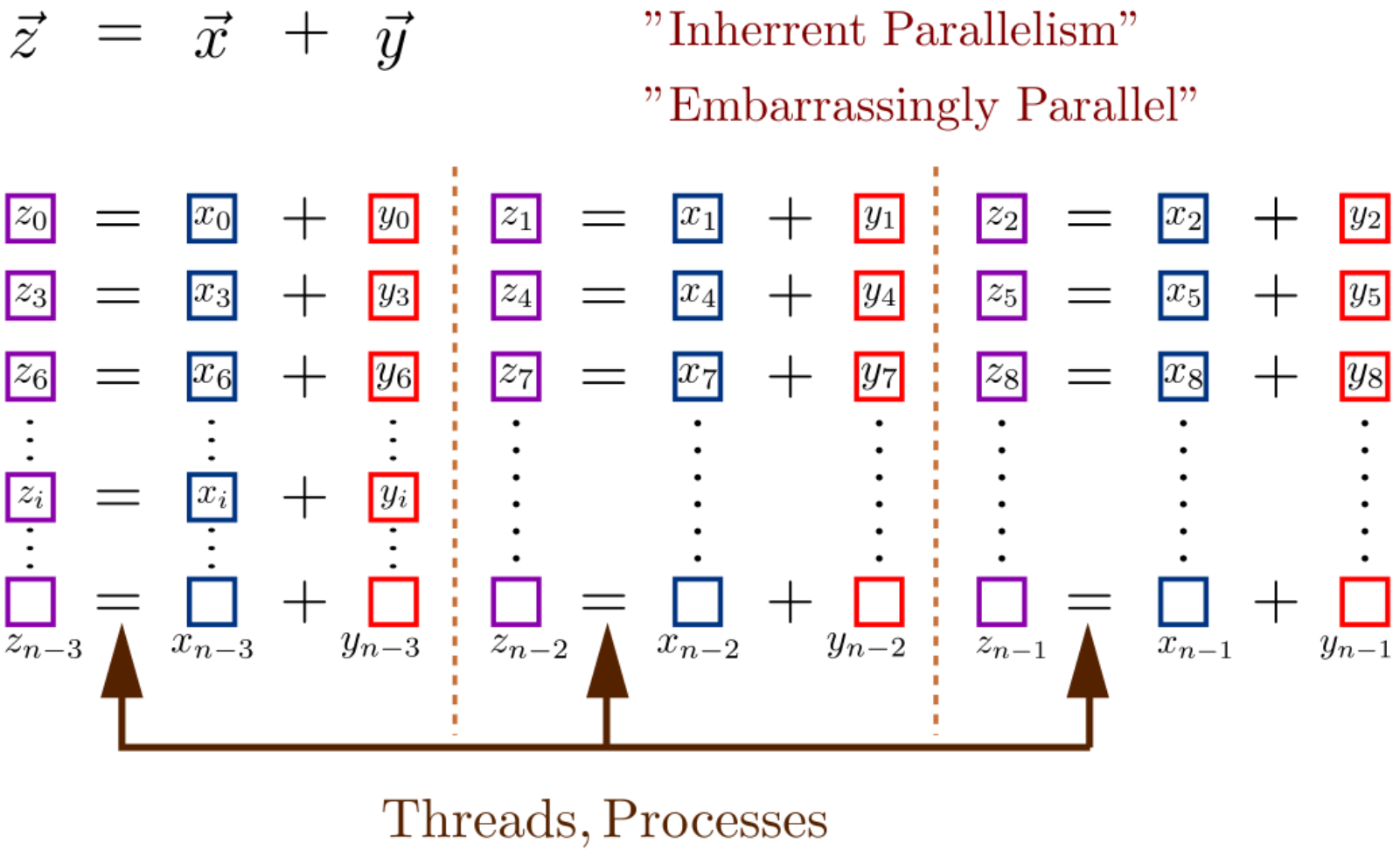
$$\vec{z} = \vec{x} + \vec{y}$$

"Inherrent Parallelism"

"Embarrassingly Parallel"

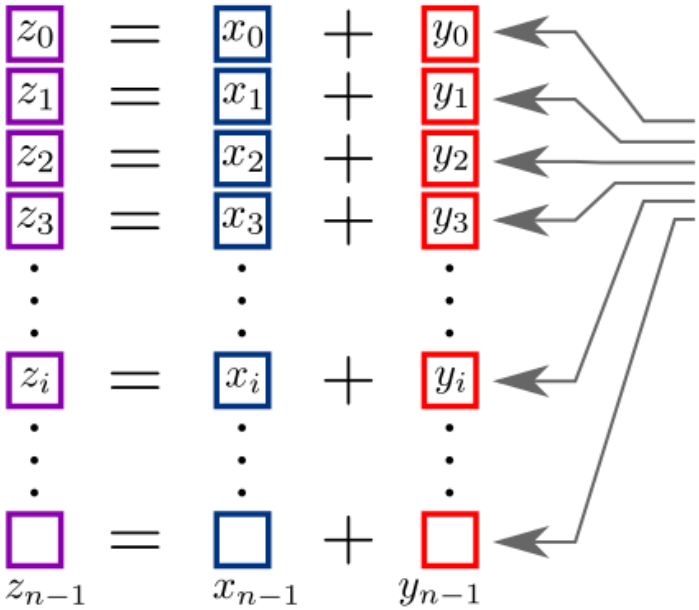


Example: Vector-Vector Addition (Overview)



Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

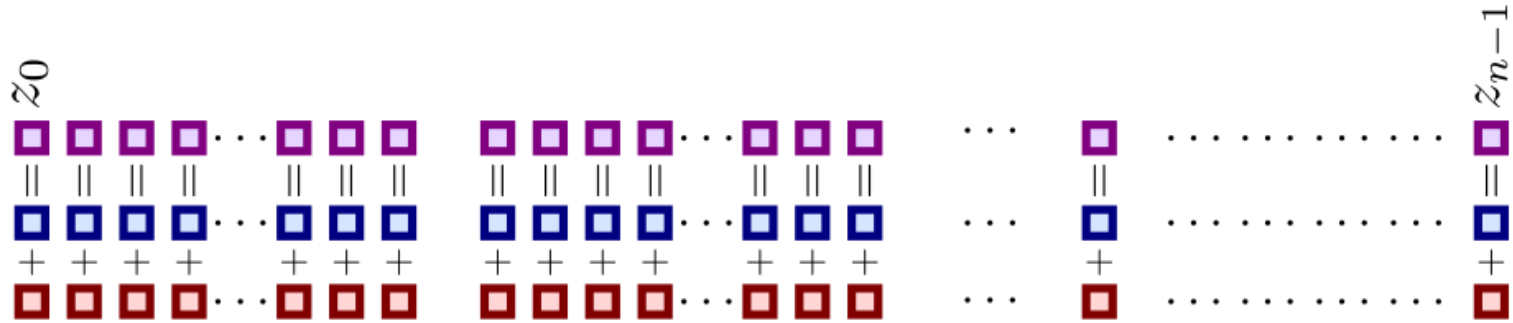


$\vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations

Example: Vector-Vector Addition (Kernel, SPMD)

```
void sum( double const* x, double const* y, double* z, int const ii )
{
  z[ ii ] = x[ ii ] + y[ ii ];
}
```

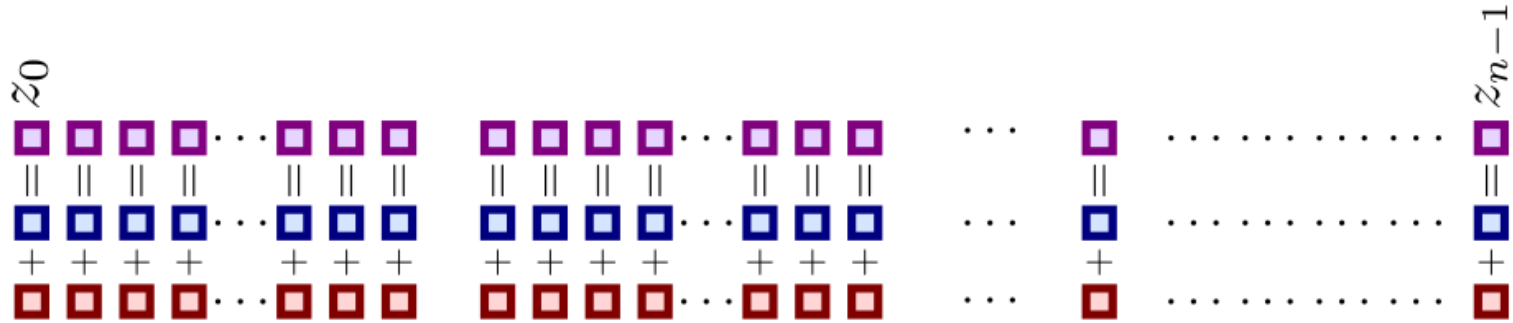


Example: Vector-Vector Addition (Kernel, SPMD)

```
void sum( double const* x, double const* y,
double* z, int const ii )
{
  z[ ii ] = x[ ii ] + y[ ii ];
}
```

"Kernel"

```
sum_kernel( x, y, z, ii)
```



Detail: OpenCL and Cuda Kernels for $\vec{z} = \vec{x} + \vec{y}$

OpenCL Kernel

```
__kernel void add_vec_kernel(
    __global double const* restrict x,
    __global double const* restrict y,
    __global double* restrict z, int const n )
{
    int const gid = get_global_id( 0 );

    if( gid < n ) z[ gid ] = x[ gid ] + y[ gid ];
}
```

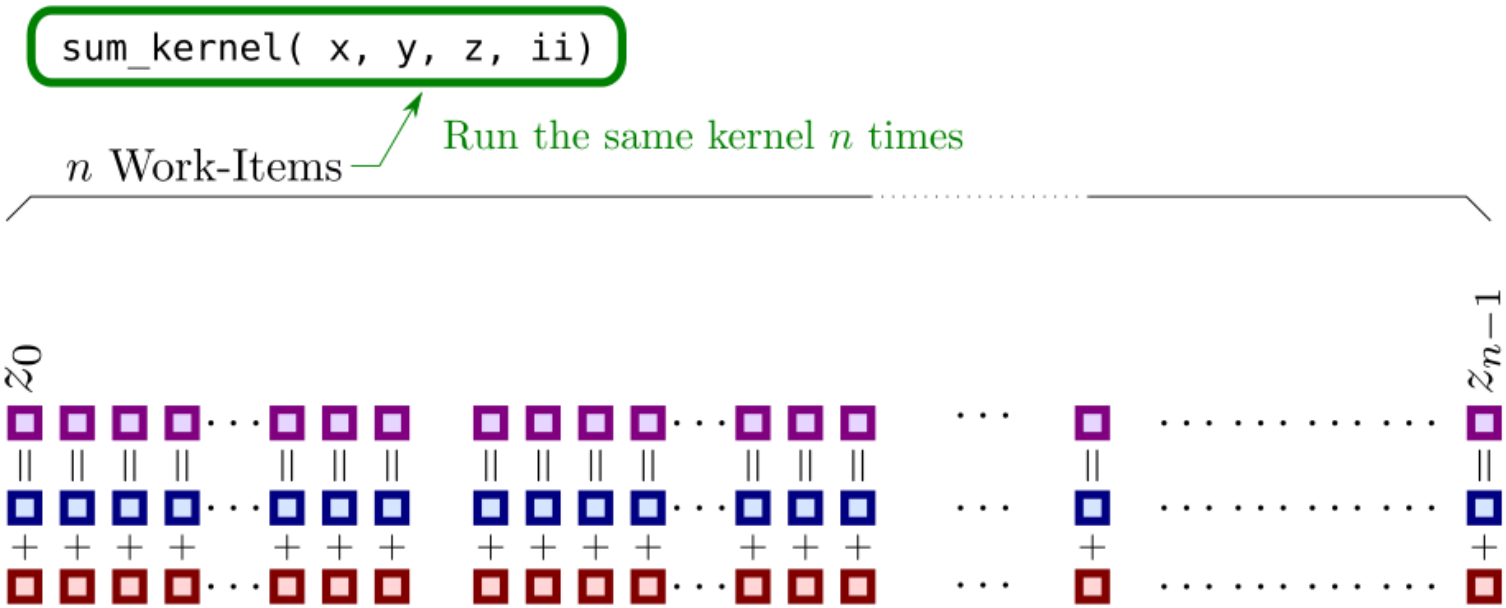
Cuda Kernel

```
__global__ void add_vec_kernel(
    double const* __restrict__ x,
    double const* __restrict__ y,
    double* __restrict__ z, int const n )
{
    int const gid = blockIdx.x * blockDim.x + threadIdx.x;

    if( gid < n ) z[ gid ] = x[ gid ] + y[ gid ];
}
```

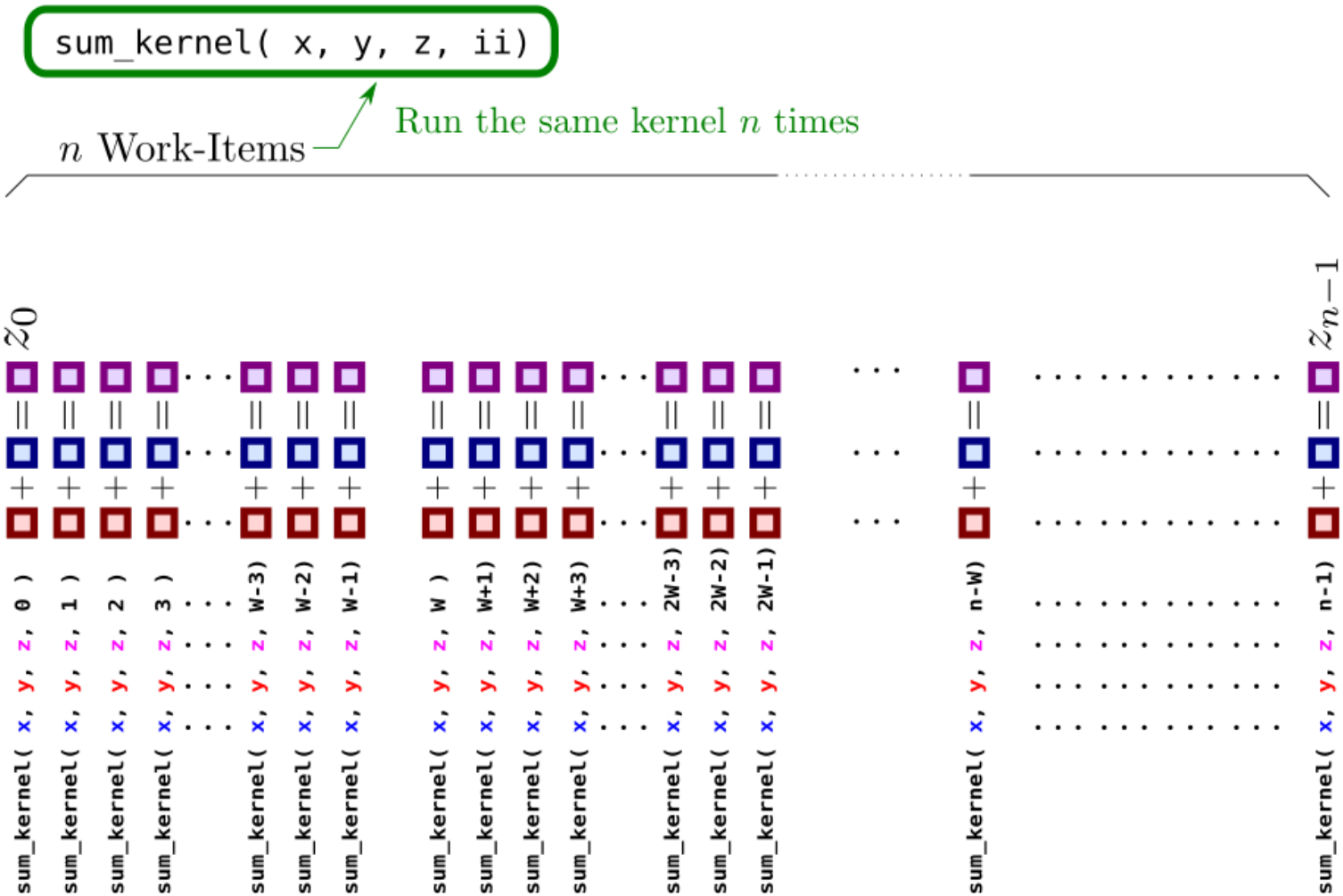
OpenCL: Building the Program, Getting the Kernel

```
/* ------------------------------------------------------------------------ */
/* Build the program and get the kernel: */

cl::Context context( device );
cl::CommandQueue queue( context, device );

std::string const kernel_source =
"__kernel void add_vec_kernel( \r\n"
    " __global double const* restrict x, __global double const* restrict y,"
    " __global double* restrict z, int const n )\r\n"
"{\r\n"
"   int const gid = get_global_id( 0 );\r\n"
"   if( gid < n ) {\r\n"
"       z[ gid ] = x[ gid ] + y[ gid ]; \r\n"
"   }\r\n"
"}\r\n";
std::string const compile_options = "-w -Werror";

cl::Program program( context, kernel_source );
cl_int ret = program.build( compile_options.c_str() );

assert( ret == CL_SUCCESS );
cl::Kernel kernel( program, "add_vec_kernel" );
```

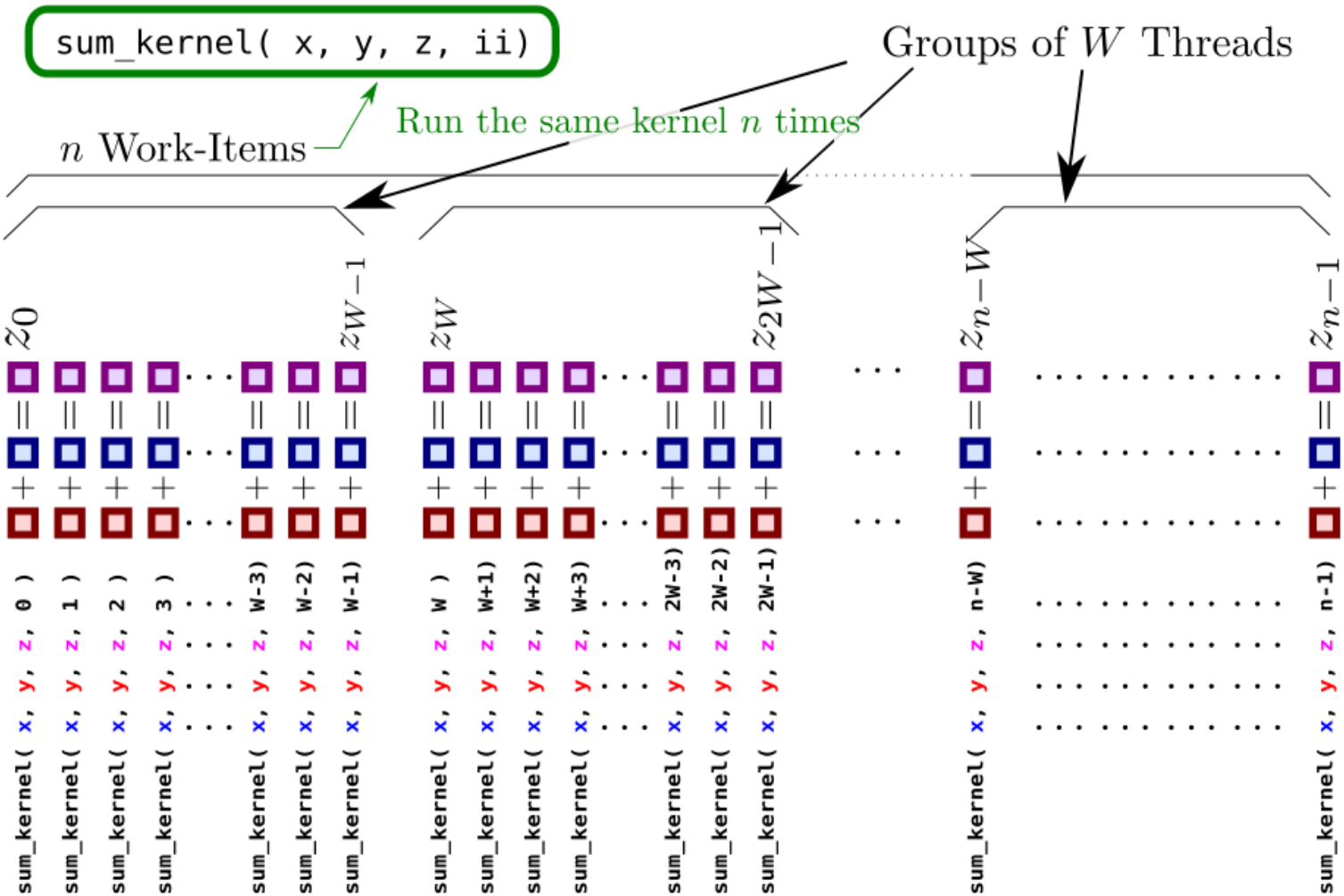
Example: Vector-Vector Addition (Kernel, SPMD)



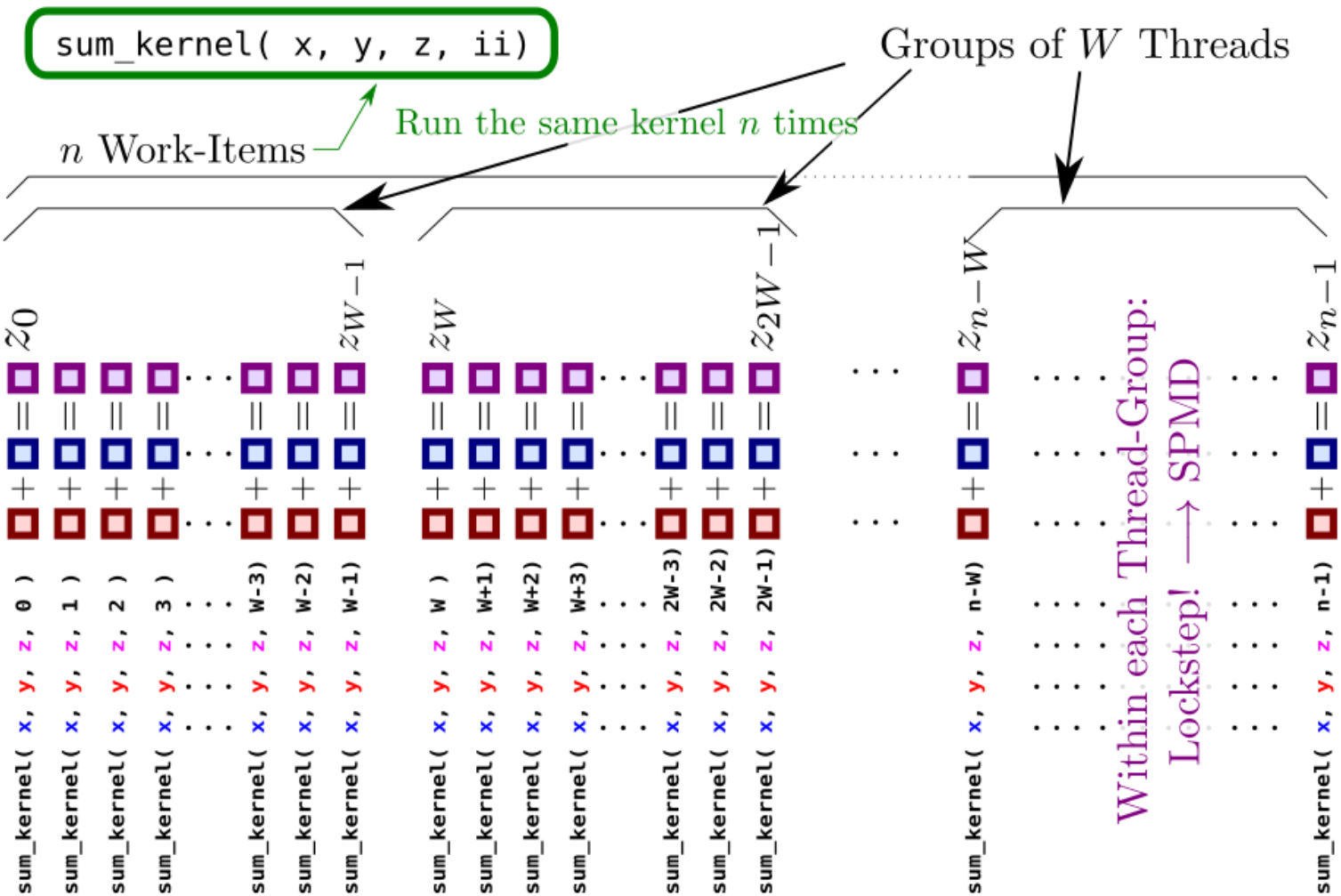
Example: Vector-Vector Addition (Kernel, SPMD)



Example: Vector-Vector Addition (Kernel, SPMD)



Example: Vector-Vector Addition (Kernel, SPMD)



Example: Vector-Vector Addition (SPMD,Teams,Grid)

- Internally: Groups of W Threads operating in lock-step
  - NVidia: Warp, $W \sim 32$
  - AMD: Wavefront, $W \sim 64$
  - Other: Team, ...
- For the user, threads are organized in a 3D geometrical structure
  - Cuda: Grid (Blocks, Threads)
  - OpenCL: Work-groups of Threads in a 3D grid
- The 3D grid is motivated by image (video) applications
- Flexible $\rightarrow$ can be treated like a 1D grid

Example: Vector-Vector Addition (SPMD,Teams,Grid)

- Internally: Groups of W Threads operating in lock-step
  - NVidia: Warp, $W \sim 32$
  - AMD: Wavefront, $W \sim 64$
  - Other: Team, ...
- For the user, threads are organized in a 3D geometrical structure
  - Cuda: Grid (Blocks, Threads)
  - OpenCL: Work-groups of Threads in a 3D grid
- The 3D grid is motivated by image (video) applications
- Flexible $\rightarrow$ can be treated like a 1D grid

Example: Vector-Vector Addition (SPMD,Teams,Grid)

- Internally: Groups of W Threads operating in lock-step
  - NVidia: Warp, $W \sim 32$
  - AMD: Wavefront, $W \sim 64$
  - Other: Team, ...
- For the user, threads are organized in a 3D geometrical structure
  - Cuda: Grid (Blocks, Threads)
  - OpenCL: Work-groups of Threads in a 3D grid
- The 3D grid is motivated by image (video) applications
- Flexible $\rightarrow$ can be treated like a 1D grid

Example: Vector-Vector Addition (SPMD,Teams,Grid)

- Internally: Groups of W Threads operating in lock-step
  - NVidia: Warp, $W \sim 32$
  - AMD: Wavefront, $W \sim 64$
  - Other: Team, ...
- For the user, threads are organized in a 3D geometrical structure
  - Cuda: Grid (Blocks, Threads)
  - OpenCL: Work-groups of Threads in a 3D grid
- The 3D grid is motivated by image (video) applications
- Flexible $\rightarrow$ can be treated like a 1D grid

Example: Vector-Vector Addition (SPMD,Teams,Grid)

- Internally: Groups of W Threads operating in lock-step
  - NVidia: Warp, $W \sim 32$
  - AMD: Wavefront, $W \sim 64$
  - Other: Team, ...
- For the user, threads are organized in a 3D geometrical structure
  - Cuda: Grid (Blocks, Threads)
  - OpenCL: Work-groups of Threads in a 3D grid
- The 3D grid is motivated by image (video) applications
- Flexible $\rightarrow$ can be treated like a 1D grid

Example: Vector-Vector Addition (SPMD,Teams,Grid)

- Internally: Groups of W Threads operating in lock-step
  - NVidia: Warp, $W \sim 32$
  - AMD: Wavefront, $W \sim 64$
  - Other: Team, ...
- For the user, threads are organized in a 3D geometrical structure
  - Cuda: Grid (Blocks, Threads)
  - OpenCL: Work-groups of Threads in a 3D grid
- The 3D grid is motivated by image (video) applications
- Flexible $\rightarrow$ can be treated like a 1D grid

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
       double const c, double const a, double const* x,
       double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
       double const c, double const a, double const* x,
       double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

W Threads

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

work-item ii

```
void c_times_a_plus_abs_xy(
        double const c, double const a, double const* x,
        double const* y, double* z, int const ii )
{
    z[ ii ] = a;

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

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

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

W Threads

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
        double const c, double const a, double const* x,
        double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

W Threads

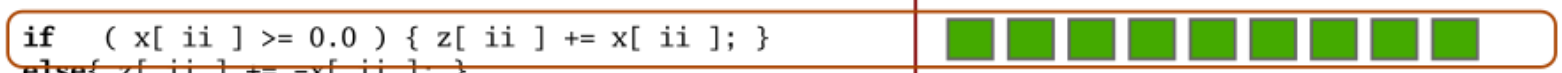
Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
       double const c, double const a, double const* x,
       double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

W Threads



Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
       double const c, double const a, double const* x,
       double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

W Threads

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
        double const c, double const a, double const* x,
        double const* y, double* z, int const ii )
{
    z[ ii ] = a;

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

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

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

W Threads

Thread Divergence

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
       double const c, double const a, double const* x,
       double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

W Threads

Thread Divergence

$\Longrightarrow$ Deferring Threads

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
       double const c, double const a, double const* x,
       double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

W Threads

Example: Vector-Vector Addition (SPMD, Branching)

One Consequence of SPMD: Branching can be Expensive

Example: Branching

```
void c_times_a_plus_abs_xy(
        double const c, double const a, double const* x,
        double const* y, double* z, int const ii )
{
   z[ ii ] = a;

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

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

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

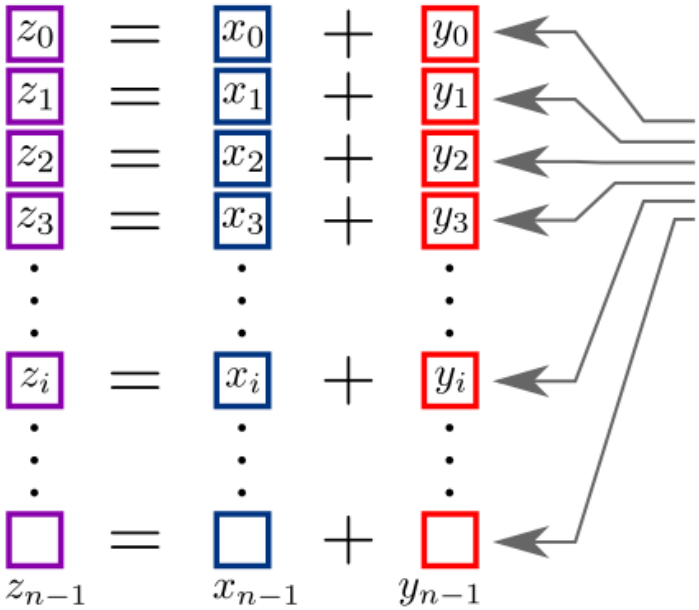
Improvements

- Branchless versions of abs() etc.
- Reorder or shuffle Data
- Framework Support

$\Longrightarrow$ Benchmark!!!

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

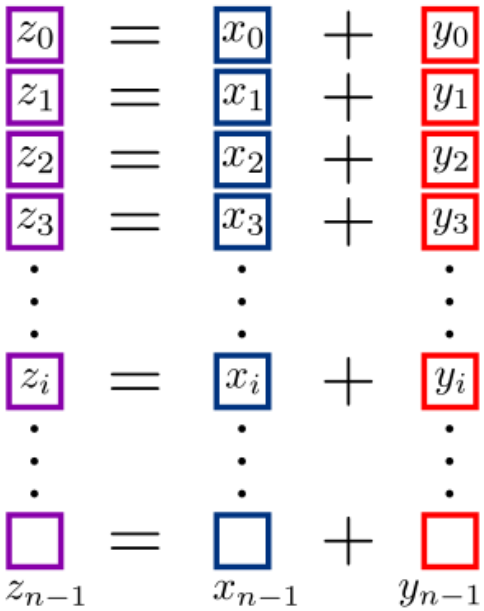


$\vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

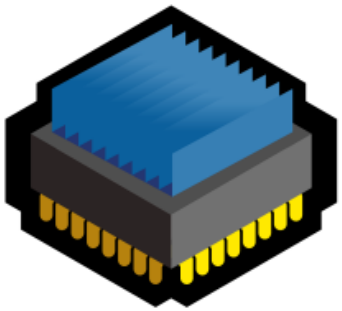


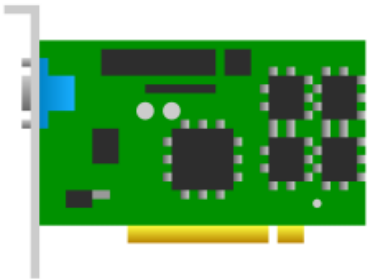
$\vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations
- No / Limited Branching

Example: Vector-Vector Addition (Device Handling)

Example: Vector-Vector Addition (Device Handling)





Host

Device

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

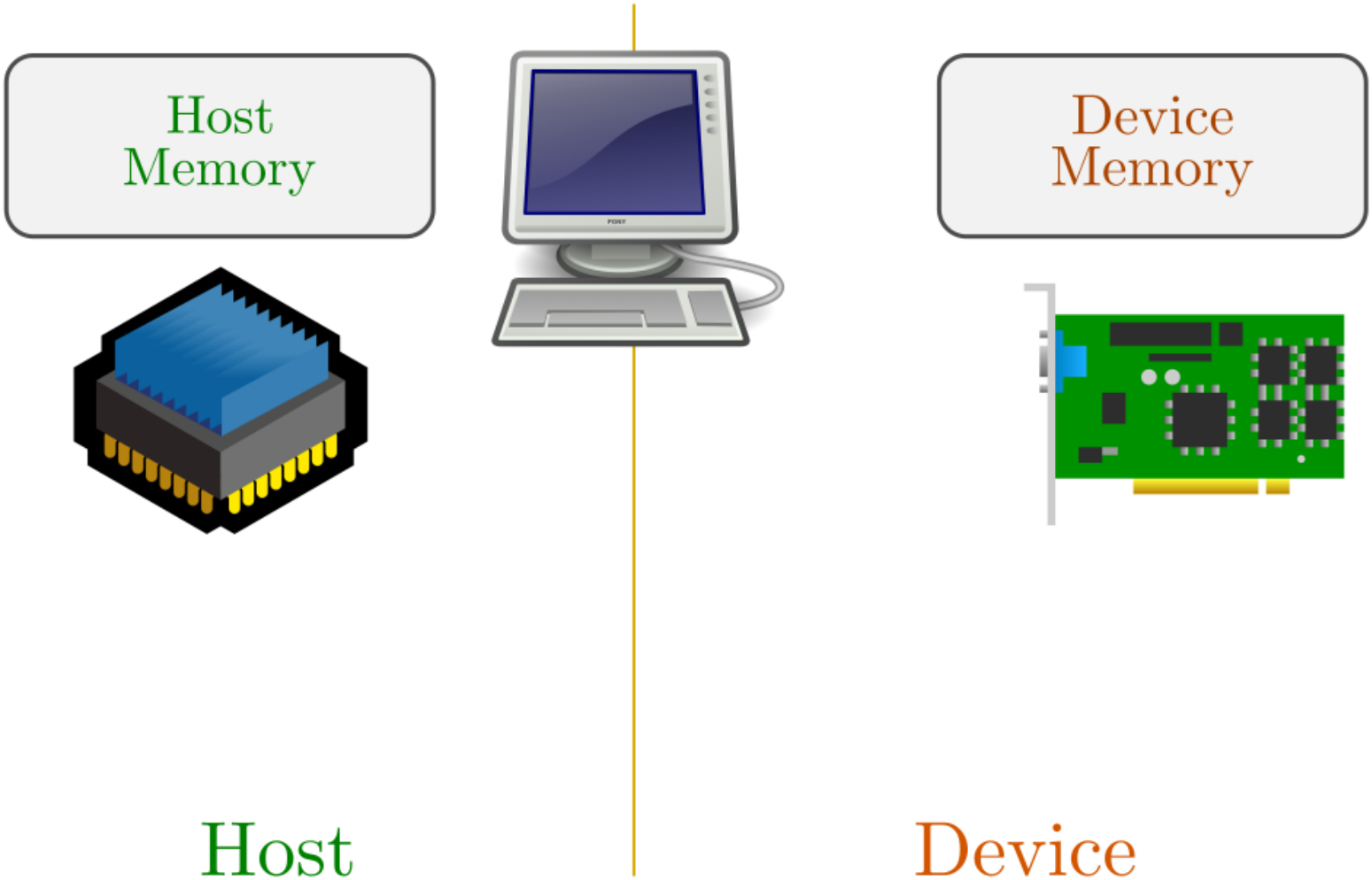
OpenCL: On the Host: prepare Device, Context, Queue

```
/* ------------------------------------------------------------------------ */
/* Select the first device on the first platform: */

std::vector< cl::Platform > platforms;
cl::Platform::get( &platforms );
std::vector< cl::Device > devices;

bool device_found = false;
cl::Device device;

for( auto const& available_platform : platforms )
{
    devices.clear();
    available_platform.getDevices( CL_DEVICE_TYPE_ALL, &devices );

    if( !devices.empty() )
    {
        device = devices.front();
        device_found = true;
        break;
    }
}
assert( device_found );

/* ------------------------------------------------------------------------ */
/* Build the program and get the kernel: */

cl::Context context( device );
cl::CommandQueue queue( context, device );
```

Cuda: Just checking ....

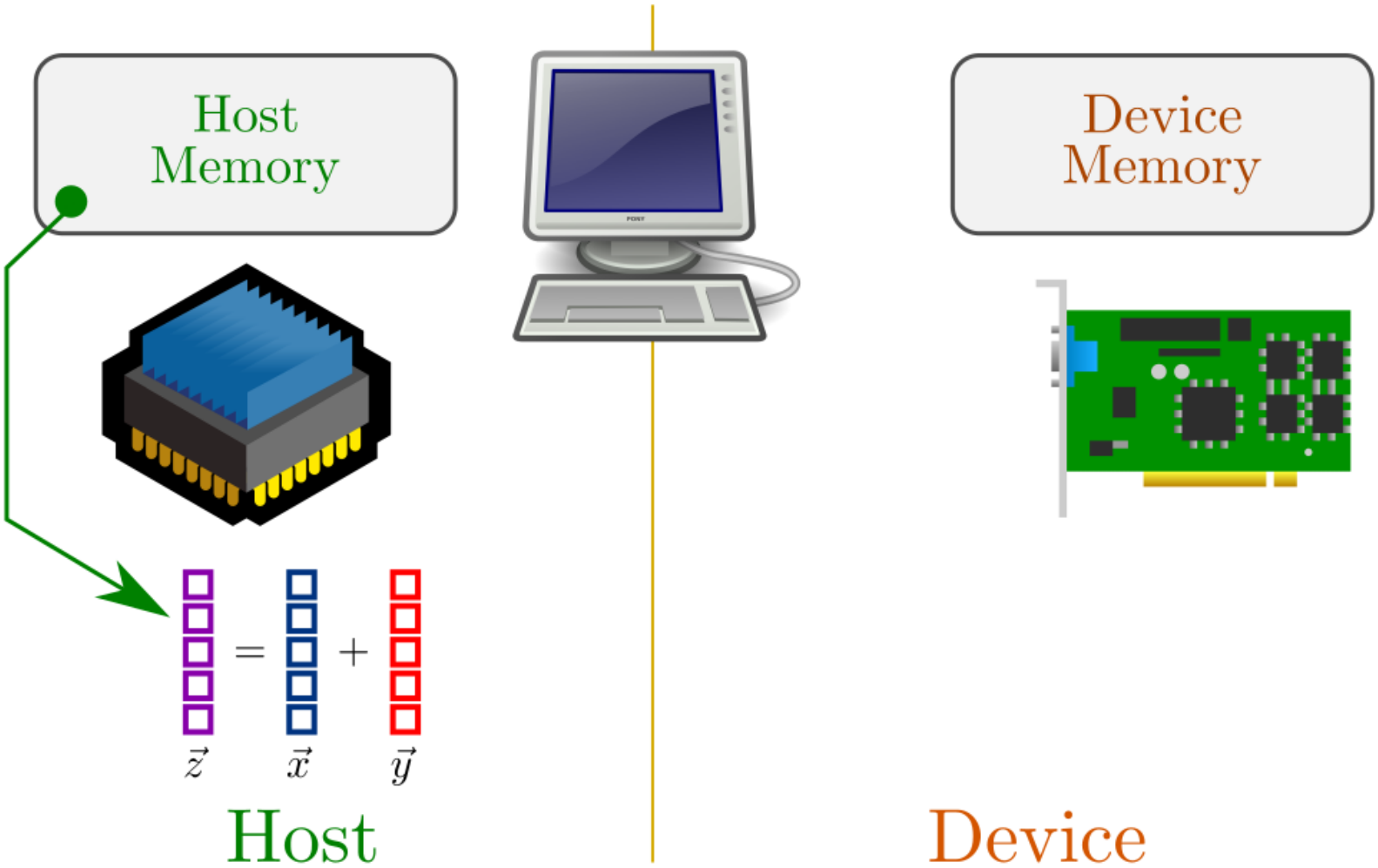
```
/* ------------------------------------------------------------------------ */
/* use the "default" / "first" Cuda device for the program: */

int device = int{ 0 };
::cudaGetDevice( &device );
cudaError_t cu_err = ::cudaDeviceSynchronize();
assert( cu_err == ::cudaSuccess );
```

Example: Vector-Vector Addition (Memory Handling)



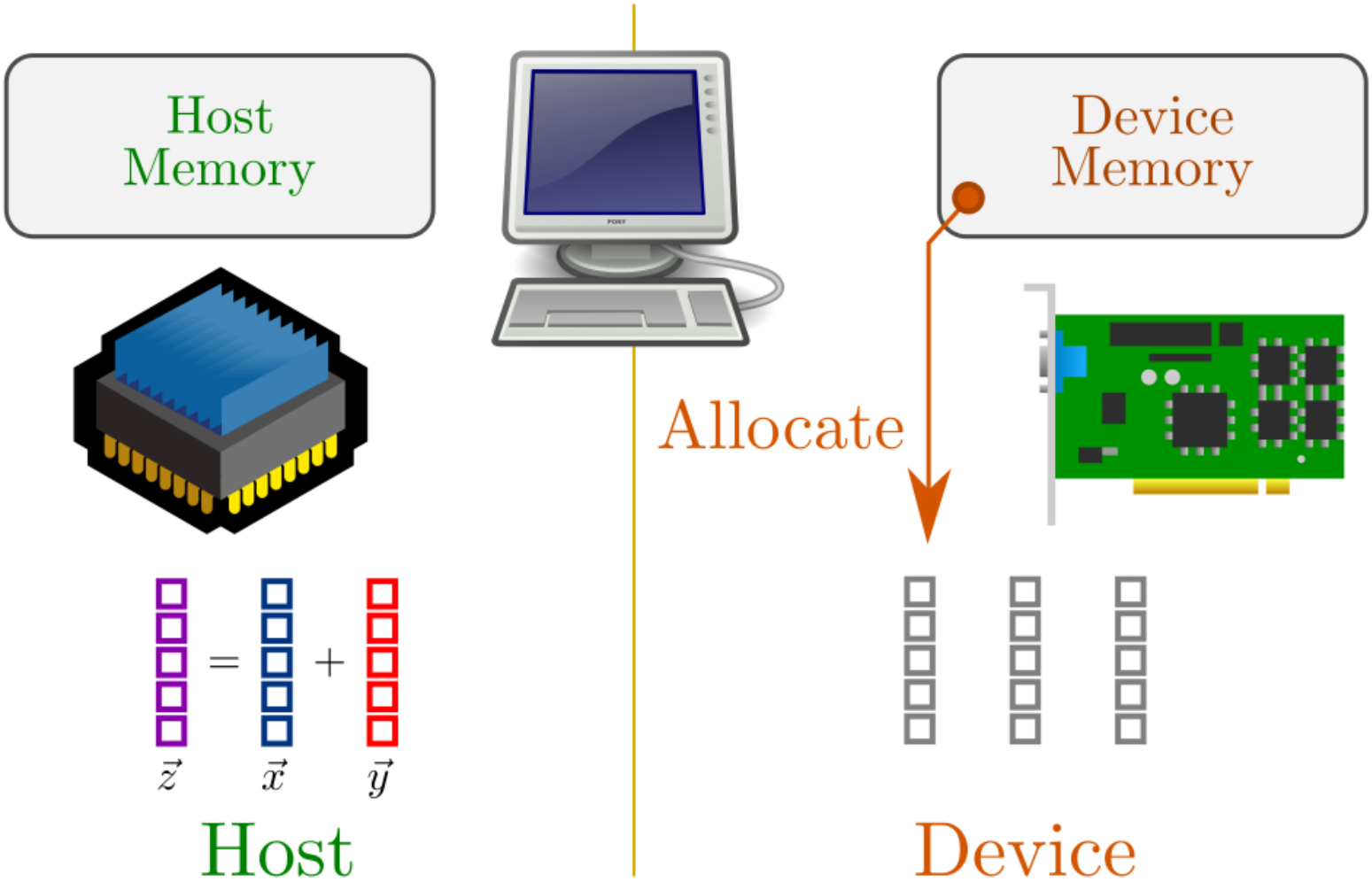
Icons: https://openclipart.org - License: Public Domain

Example: Vector-Vector Addition (Memory Handling)



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

Example: Vector-Vector Addition (Memory Handling)



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

Allocating the Buffers on the Device

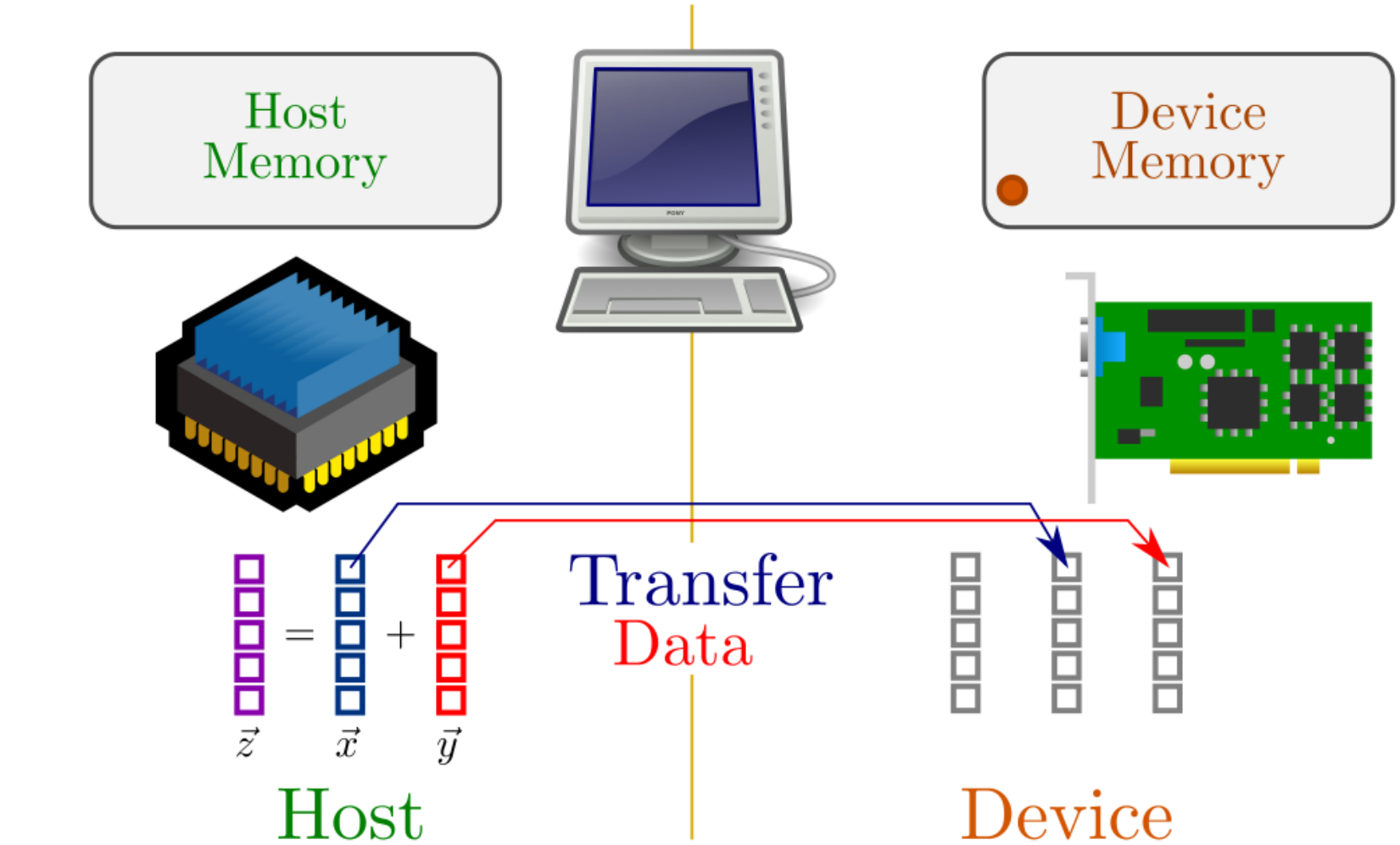
OpenCL:

```
/* ------------------------------------------------------------------ */
/* Allocate the buffers on the device */
/* x_arg, y_arg, z_arg ... handles on the host side managing buffers in *
* the device memory */

cl::Buffer x_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
cl::Buffer y_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
cl::Buffer z_arg( context, CL_MEM_READ_WRITE, sizeof( double ) * N, nullptr );
```

Cuda:

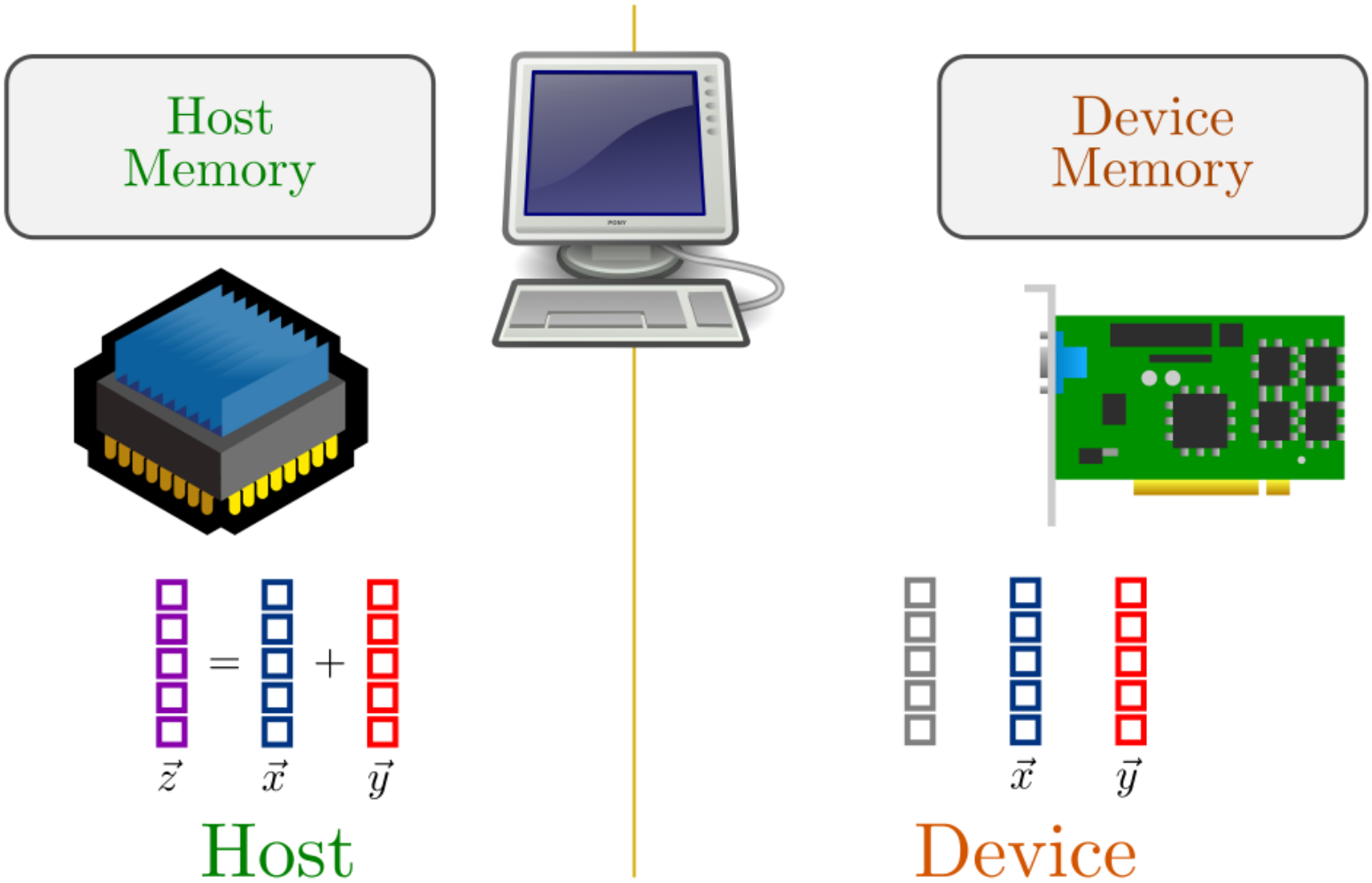
```
/* ------------------------------------------------------------------ */
/* Allocate the buffers on the device */
/* x_arg, y_arg, z_arg ... handles on the host side managing buffers in *
* the device memory */

double* x_arg = nullptr;
double* y_arg = nullptr;
double* z_arg = nullptr;

::cudaMalloc( &x_arg, sizeof( double ) * N );
::cudaMalloc( &y_arg, sizeof( double ) * N );
::cudaMalloc( &z_arg, sizeof( double ) * N );
```

Example: Vector-Vector Addition (Memory Handling)



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

Example: Vector-Vector Addition (Memory Handling)



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

Transfering $\vec{x}$ and $\vec{y}$ to the Device Memory

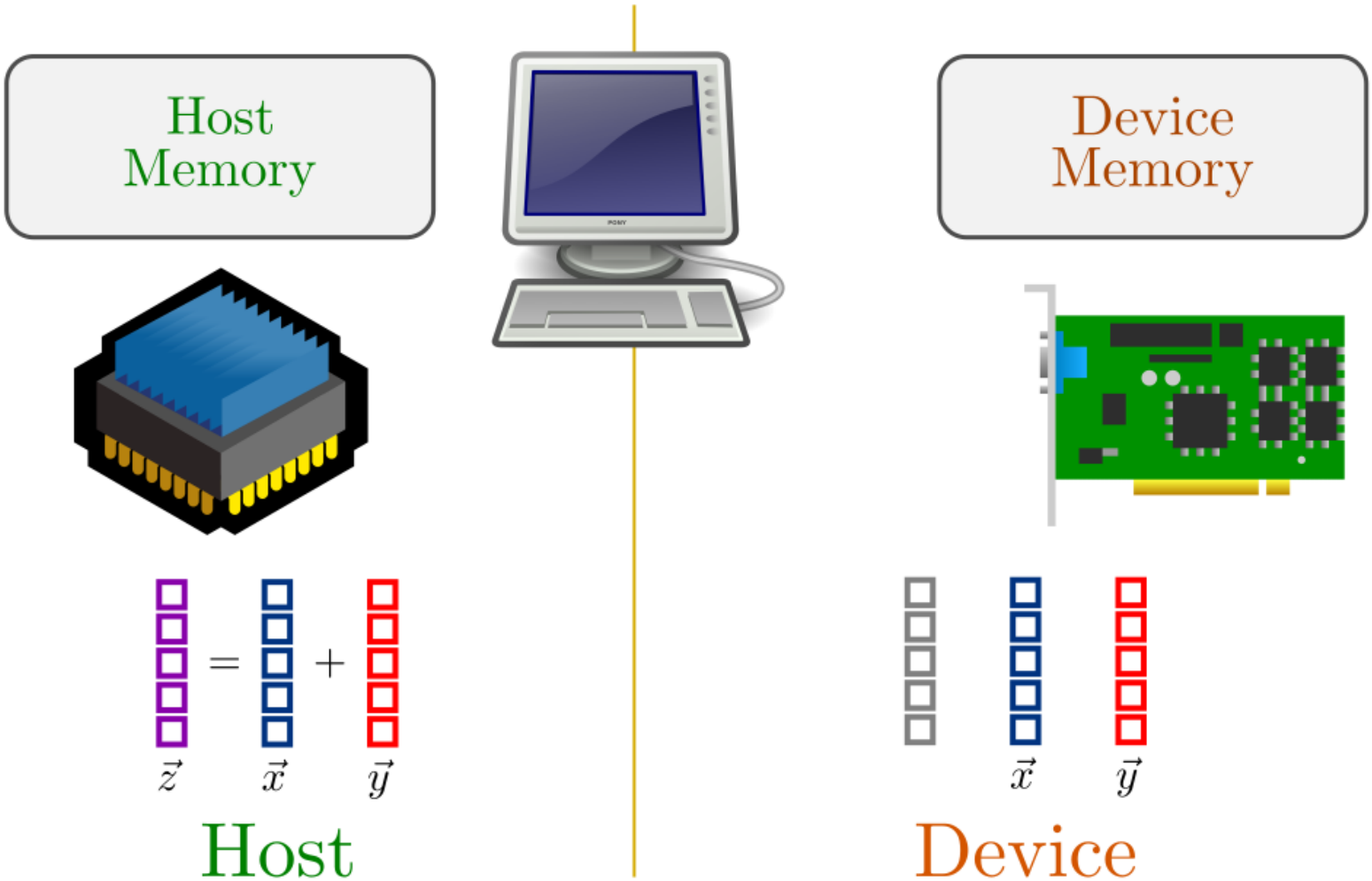
OpenCL:

```
/* Transfer x and y from the host to the device */

ret = queue.enqueueWriteBuffer( x_arg, CL_TRUE, std::size_t{ 0 },
                                x.size() * sizeof( double ), x.data() );
assert( ret == CL_SUCCESS );

ret = queue.enqueueWriteBuffer( y_arg, CL_TRUE, std::size_t{ 0 },
                                y.size() * sizeof( double ), y.data() );
assert( ret == CL_SUCCESS );
```

Cuda:

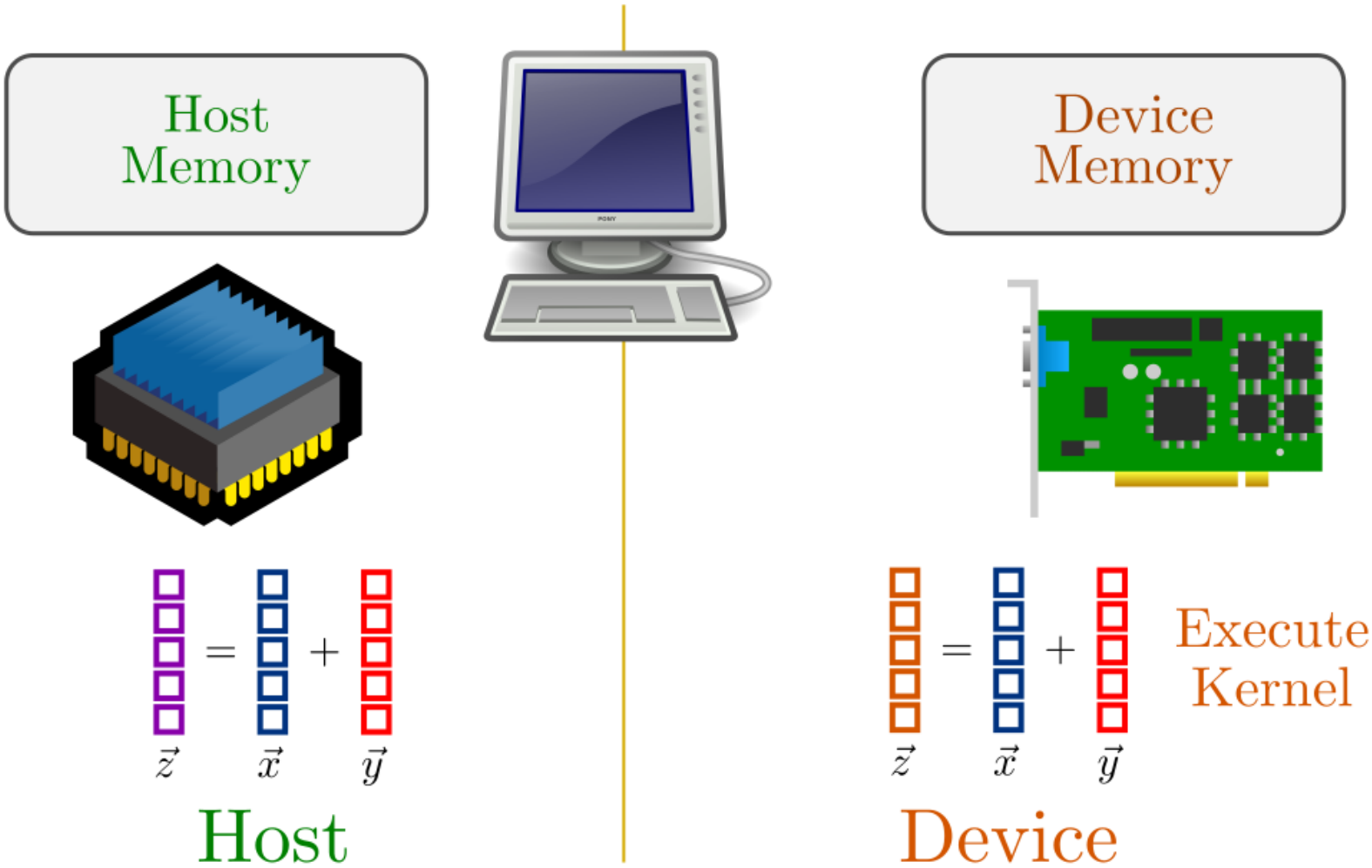
```
/* ------------------------------------------------------------------- */
/* Transfer x and y from host to device */

::cudaMemcpy( x_arg, x.data(), sizeof( double ) * N, cudaMemcpyHostToDevice );
::cudaMemcpy( y_arg, y.data(), sizeof( double ) * N, cudaMemcpyHostToDevice );
```

Example: Vector-Vector Addition (Running the Kernel)



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

Example: Vector-Vector Addition (Running the Kernel)



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

Running the kernel

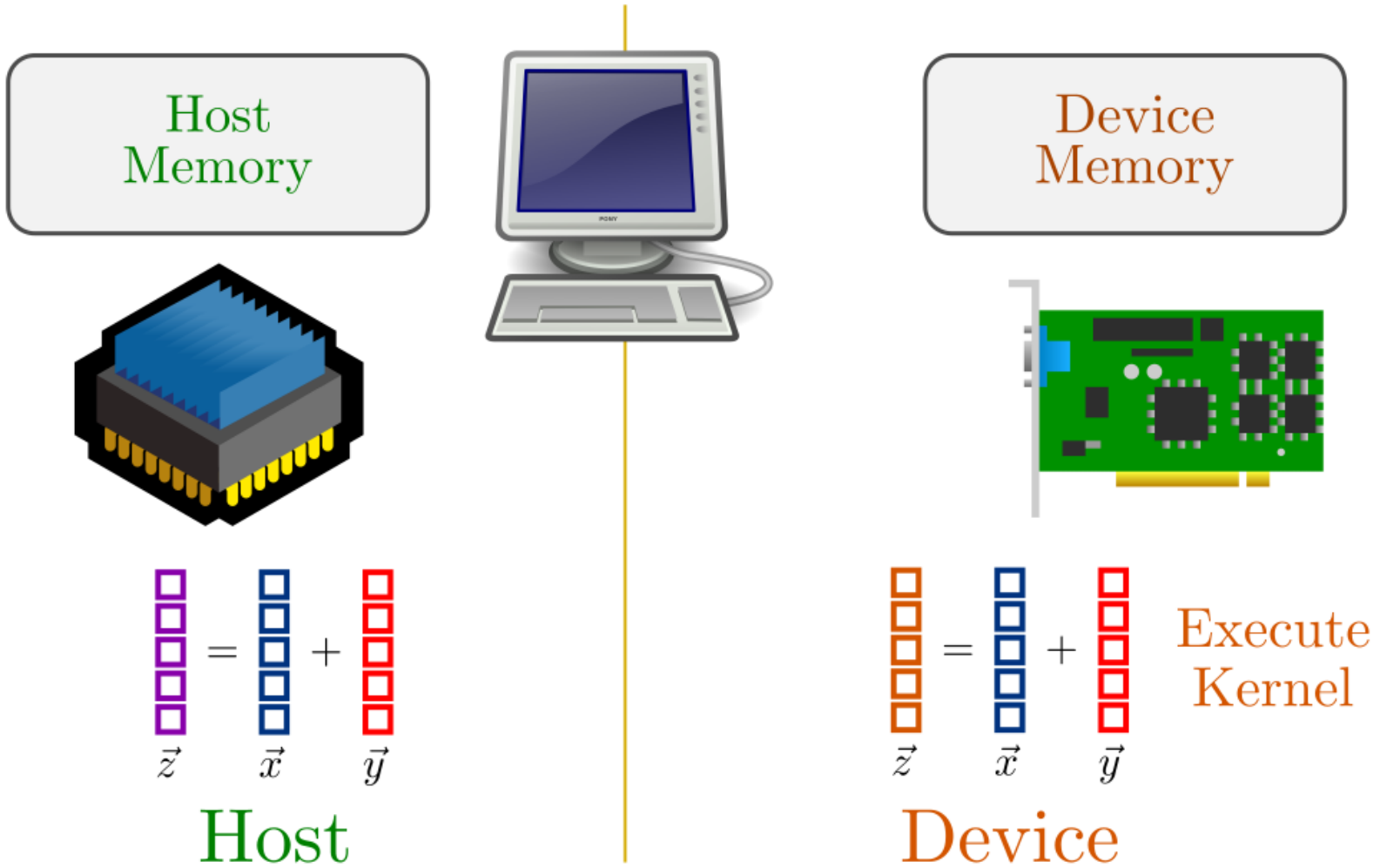
OpenCL:

```
/* ------------------------------------------------------------------------ */
/* Prepare the kernel for execution: bind the arguments to the kernel */

kernel.setArg( 0, x_arg );
kernel.setArg( 1, y_arg );
kernel.setArg( 2, z_arg );
kernel.setArg( 3, N );

/* ------------------------------------------------------------------------ */
/* execute the kernel on the device */
cl::NDRange offset = cl::NullRange; cl::NDRange local  = cl::NullRange;

ret = queue.enqueueNDRangeKernel( kernel, offset, N, local );
assert( ret == CL_SUCCESS );
```

Cuda:

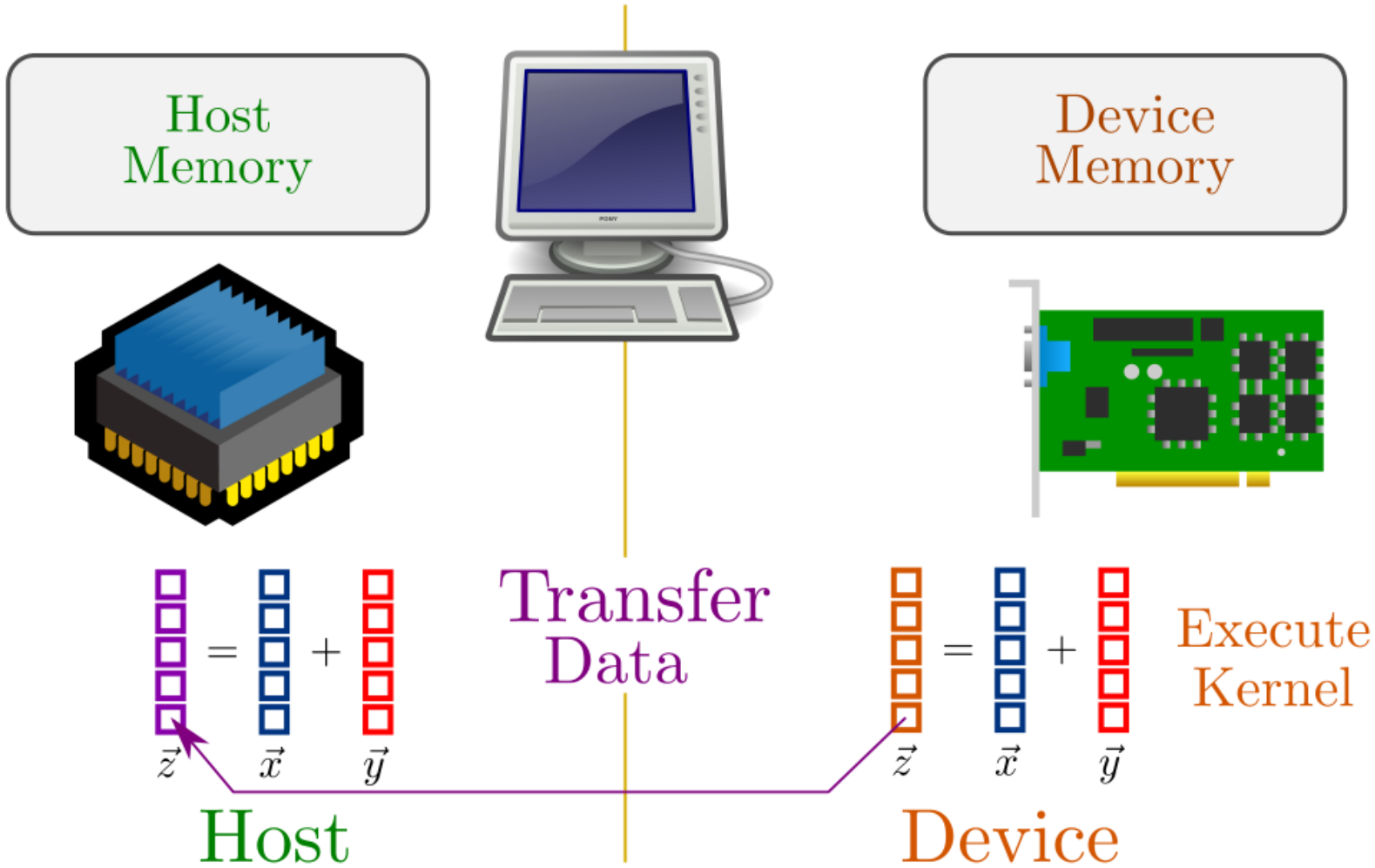
```
/* ------------------------------------------------------------------------ */
/* execute kernel on the device */

int32_t const threads_per_block = int32_t{ 128 };

int32_t const num_blocks =
    ( N + threads_per_block - int32_t{ 1 } ) / threads_per_block;

add_vec_kernel<<< num_blocks, threads_per_block >>>( x_arg, y_arg, z_arg, N );

cu_err = ::cudaPeekAtLastError();
assert( cu_err == ::cudaSuccess );
```

Example: Vector-Vector Addition (Retrieve the result)



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

Example: Vector-Vector Addition (Retrieve the result)



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

Detail: Transfering $\vec{z}$ $\vec{y}$ from the Device Memory

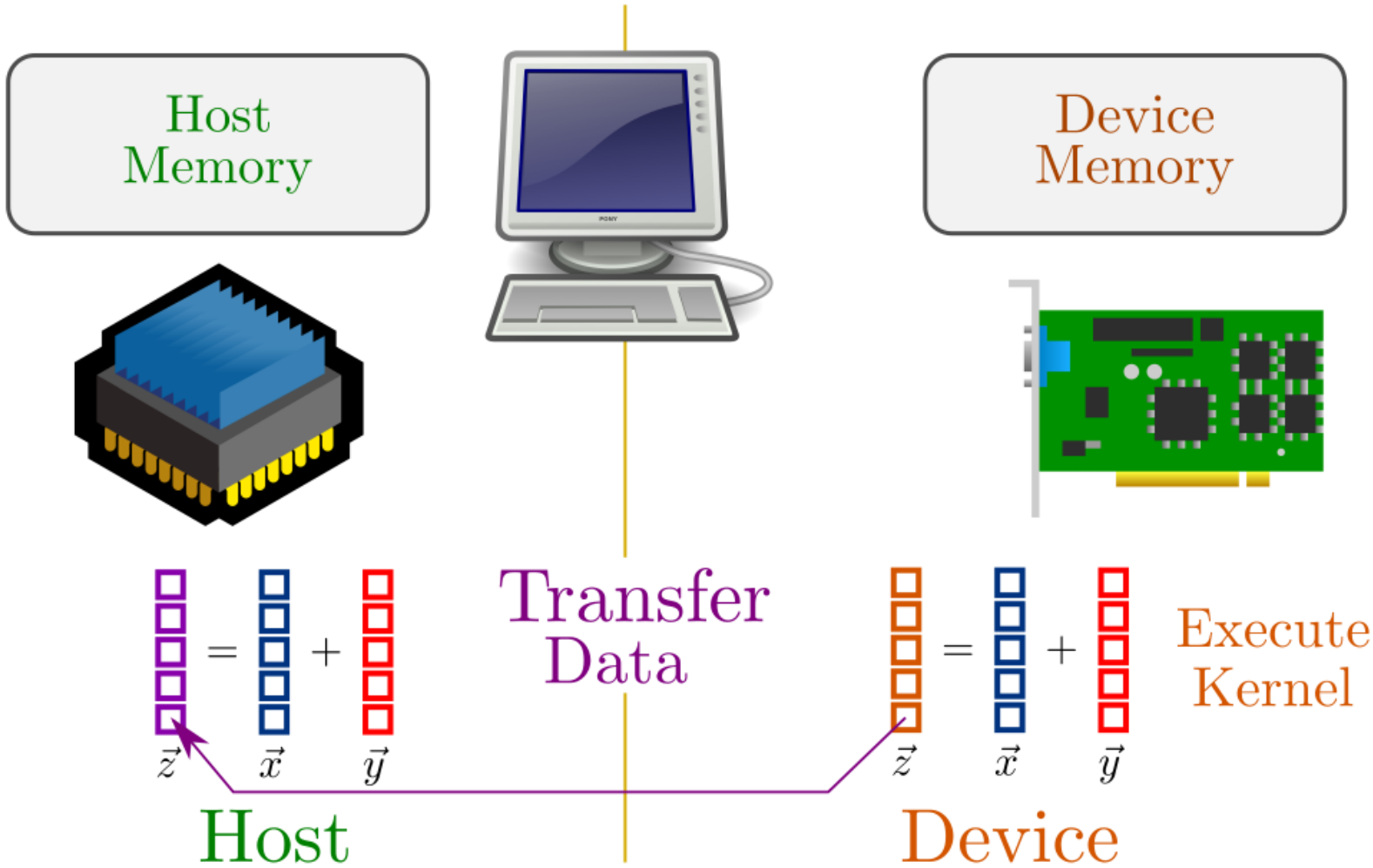
OpenCL:

```
/* transfer the result from the device buffer to the host buffer */

ret = queue.enqueueReadBuffer( z_arg, CL_TRUE, std::size_t{ 0 },
                               z.size() * sizeof( double ), z.data() );
assert( ret == CL_SUCCESS );
```

Cuda:

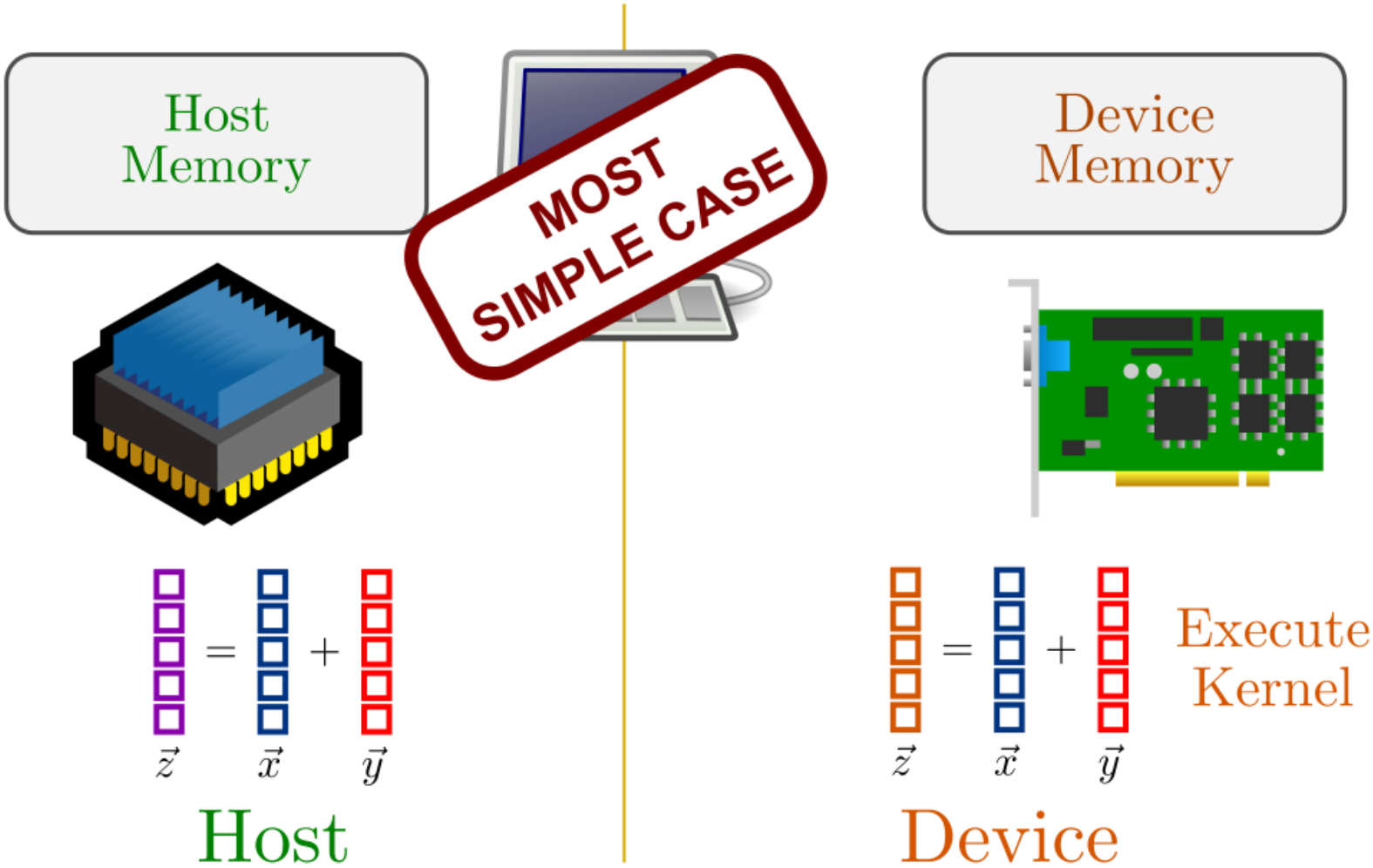
```
/* transfer the result from the device buffer to the host buffer */

::cudaMemcpy( z.data(), z_arg, sizeof( double ) * N, cudaMemcpyDeviceToHost );
cu_err = ::cudaPeekAtLastError();
assert( cu_err == ::cudaSuccess );
```

Example: Vector-Vector Addition (Memory Access Patterns)



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

Example: Vector-Vector Addition (Memory Access Patterns)



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

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

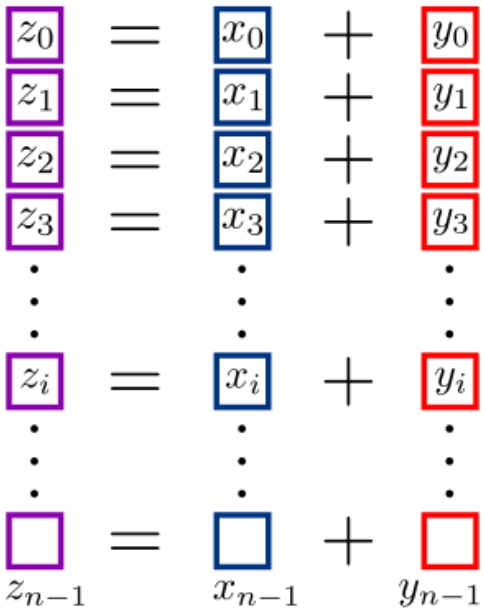
- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Memory Access Patterns)

- Device Memory: Different Memory Regions / Domains
  1. "Global Memory": Visible to all threads, $\leftrightarrow$ Host
  2. "Shared Memory": Visible to all threads in a wavefront/warp
  3. "Thread-local Memory": Visible only to each thread
  4. "Constant Memory": Visible to all threads, read-only
- 1.) - 4.) Differences in latency, bandwith and availability
- 1.) and 2.) may require Synchronization
- Further Complications/Goodies: Virtual shared memory, host-addressible memory, pinned memory, ...
- $\longrightarrow$ Trade-offs
- Illustrated Example: Allocate $\vec{x}, \vec{y}$ and $\vec{z}$ in "global" device memory
- Each element is accessed only once, no data sharing between different threads necessary

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$



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

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations
- No / Limited Branching

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

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

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

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations
- No / Limited Branching
- Memory: High Degree of Locality
- Simple Memory Access Pattern
- Number of elements n large

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

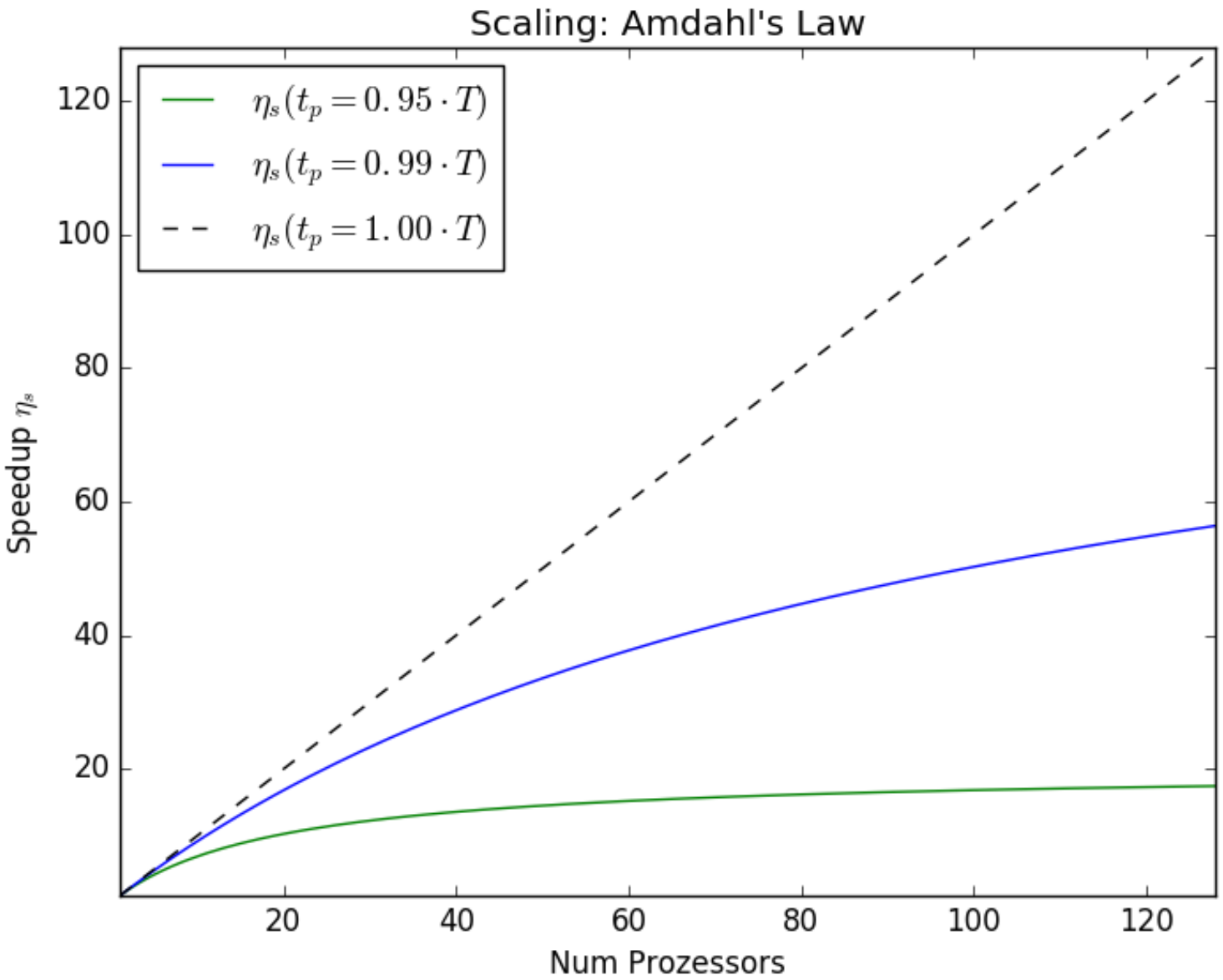
Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling

- Program with total run-time T
- Program operates on a problem of **constant size** (example: vectors of size n)
- Question: how much can we speed up the execution of the problem if parallelize it?
- Speedup $n_S = ?$
  - Amdahl's law: $\eta_S(n_p) = \frac{T}{t_s + \frac{t_p}{n_p}}$
  - t_s fraction of run-time that is "sequential" ("serial")
  - t_p fraction of run-time that can be run in "parallel" on n_p "processors"
  - $T = t_s + t_p$

Performance Analysis: Scaling



Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta(n_p) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta(n_p) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta\left(n_p\right) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta\left(n_p\right) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta\left(n_p\right) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta\left(n_p\right) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

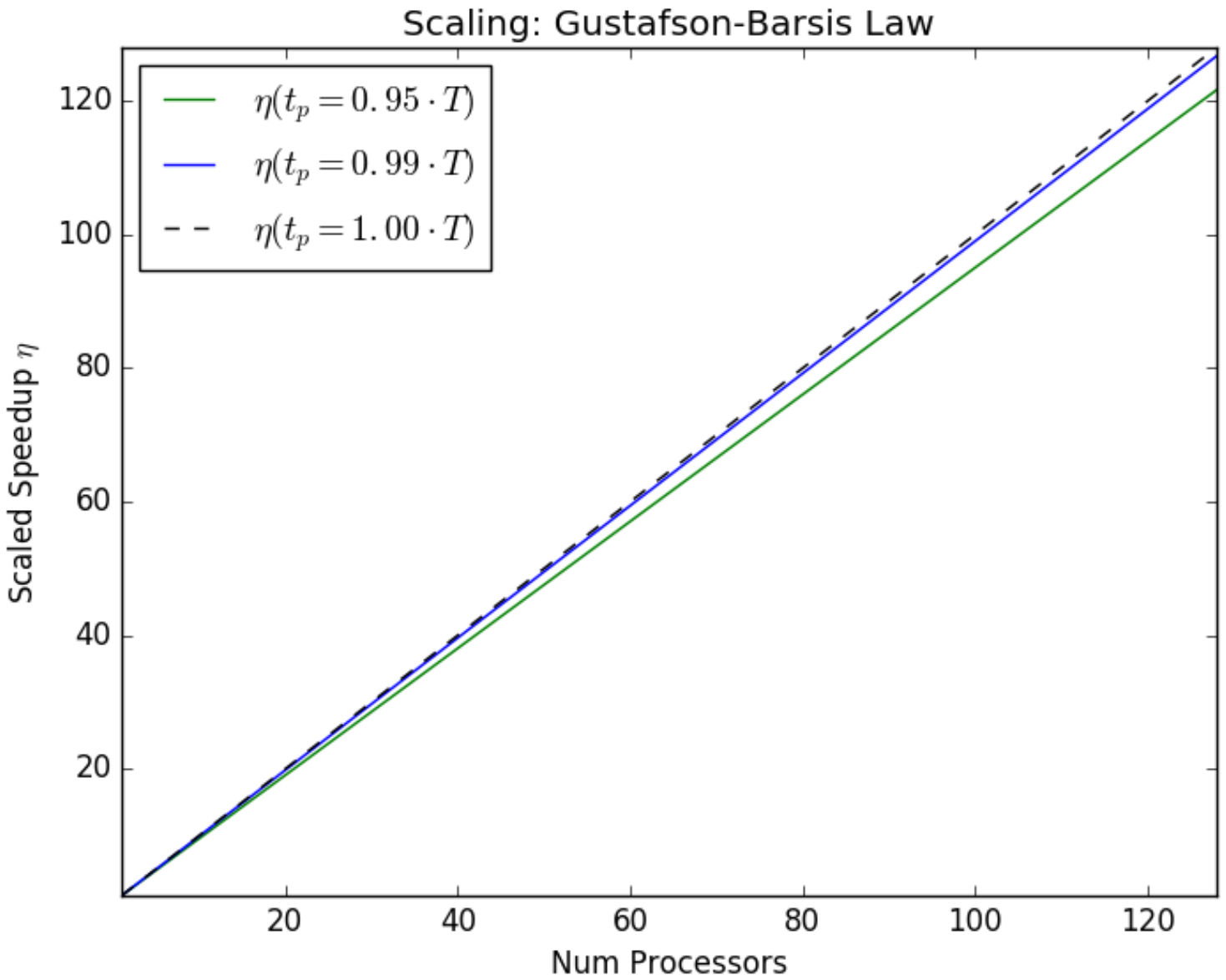
Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta\left(n_p\right) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling

- Amdahl's law is a bit pessimistic
- i.e. $t_p = 0.99 \cdot T, n_p \to 128 \Rightarrow \eta_S < 50\%$
- What if we decide to use increasing n_p to **"grow" the problem size**?
- Scaled speedup η
  - Gustafson-Barsis law: $\eta\left(n_p\right) = \frac{T}{f_s + n_p \cdot f_p}$
  - $f_s = t_s/T$ fraction of run-time that is "sequential" ("serial")
  - $t_p = t_p/T$ fraction of run-time that can be run in "parallel" on n_p "processors"
  - $1 = f_s + f_p$

Performance Analysis: Scaling



Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

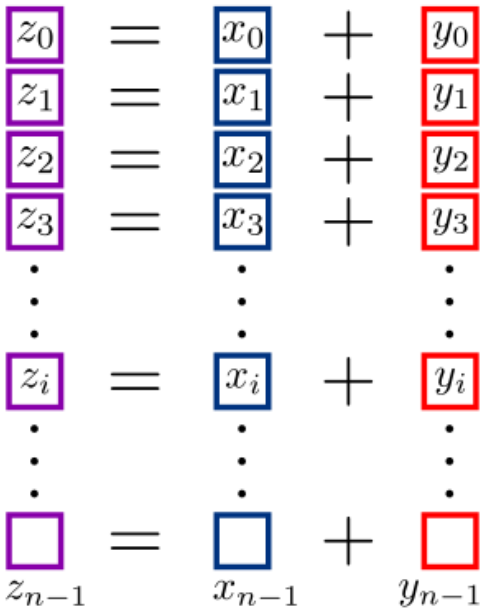
$$\begin{array}{ccccc}
z_0 & = & x_0 & + & y_0 \\
z_1 & = & x_1 & + & y_1 \\
z_2 & = & x_2 & + & y_2 \\
z_3 & = & x_3 & + & y_3 \\
\vdots & & \vdots & & \vdots \\
z_i & = & x_i & + & y_i \\
\vdots & & \vdots & & \vdots \\
\square & = & \square & + & \square \\
z_{n-1} & & x_{n-1} & & y_{n-1}
\end{array}$$

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

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations
- No / Limited Branching
- Memory: High Degree of Locality
- Simple Memory Access Pattern
- Number of elements n large

Example: Vector-Vector Addition (Overview)

$$\vec{z} = \vec{x} + \vec{y}$$

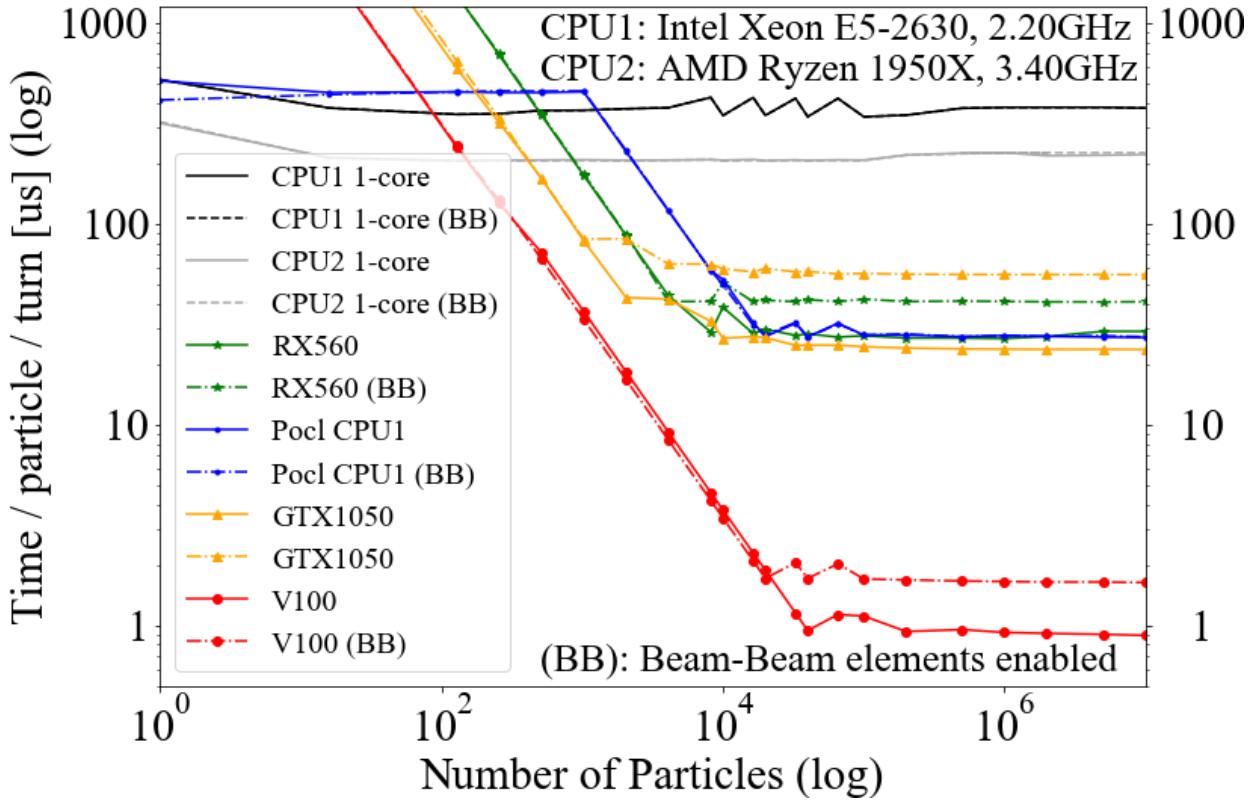


$\vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$

- Independent Operations
- Operations of comparable "cost"
- Identical Program (SPMD, SIMD)
- Arithmetic / Numerical Operations
- No / Limited Branching
- Memory: High Degree of Locality
- Simple Memory Access Pattern
- Number of elements n large

Good Match for GPU Computing

Single-Particle Tracking on GPUs? ⇒ SixTrackLib



- SixTrackLib: `https://github.com/SixTrack/sixtracklib`
- Source: De Maria et al ICAP 2018 "SixTrack project: status, running environment and new developments"
- Differences in performance can be explained using the concept of "occupancy" / "hardware utilization"

Occupancy and Hardware Utilization

- Remember: internally threads are organized in Teams of W members
- The number of teams that can scheduled to run (or run): limited by
- Hardware Resources $\leftrightarrow$ Complexity of Kernel
- More Resources required by a kernel $\rightarrow$ less Teams of W threads can be kept "in flight"
- Less teams "in-flight" $\rightarrow$ more impact of I/O latencies $\rightarrow$ larger serial fraction t_s

Occupancy and Hardware Utilization

- Remember: internally threads are organized in Teams of W members
- The number of teams that can scheduled to run (or run): limited by
- Hardware Resources $\leftrightarrow$ Complexity of Kernel
- More Resources required by a kernel $\rightarrow$ less Teams of W threads can be kept "in flight"
- Less teams "in-flight" $\rightarrow$ more impact of I/O latencies $\rightarrow$ larger serial fraction t_s

Occupancy and Hardware Utilization

- Remember: internally threads are organized in Teams of W members
- The number of teams that can scheduled to run (or run): limited by
- Hardware Resources $\leftrightarrow$ Complexity of Kernel
- More Resources required by a kernel $\rightarrow$ less Teams of W threads can be kept "in flight"
- Less teams "in-flight" $\rightarrow$ more impact of I/O latencies $\rightarrow$ larger serial fraction t_s

Occupancy and Hardware Utilization

- Remember: internally threads are organized in Teams of W members
- The number of teams that can scheduled to run (or run): limited by
- Hardware Resources $\leftrightarrow$ Complexity of Kernel
- More Resources required by a kernel $\rightarrow$ less Teams of W threads can be kept "in flight"
- Less teams "in-flight" $\rightarrow$ more impact of I/O latencies $\rightarrow$ larger serial fraction t_s

Occupancy and Hardware Utilization

- Remember: internally threads are organized in Teams of W members
- The number of teams that can scheduled to run (or run): limited by
- Hardware Resources $\leftrightarrow$ Complexity of Kernel
- More Resources required by a kernel $\rightarrow$ less Teams of W threads can be kept "in flight"
- Less teams "in-flight" $\rightarrow$ more impact of I/O latencies $\rightarrow$ larger serial fraction t_s

Extra Slides

pyopencl (complete example)

```
import numpy as np; import pyopencl as cl
a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags; RO= mf.READ_ONLY | mf.COPY_HOST_PTR
a_g = cl.Buffer(ctx, RO, hostbuf=a_np)
b_g = cl.Buffer(ctx, RO, hostbuf=b_np)
prg = cl.Program(ctx, """ <opencl as string """).build()
res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
prg.sum(queue, a_np.shape, None, a_g, b_g, res_g)
res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)
```

GPU at CERN

- ABP server (4x NVidia V100)
- TechLab servers (P100, GTX 1080, AMD W8100) booking needed
- HTCondor servers from next year (16x V100)

Best GPU for DP workloads

Name	Type	Year	DP [GFlops]	RAM [GB]	Price [$ 2018]
Nvidia Tesla V100	Server	2017	7450	32	6000
Nvidia Quadro GV100	Workstation	2017	7400	32	9000
AMD Radeon Instinct MI60	Server	2019	7400	32	?
AMD Titan V	Workstation	2018	6144	12	3000
Nvidia Quadro GP100	Workstation	2016	5168	16	9000
Nvidia Tesla P100	Server	2016	4760	16	6000
AMD FirePro S9170	Server	2014	2620	32	3500
AMD FirePro W9100	Workstation	2014	2619	16	2200
AMD FirePro S9150	Server	2014	2530	16	2000
AMD FirePro W8100	Workstation	2014	2109	16	900)