Concurrent Programming

C#
Hardware Acceleration

Resources

- https://ilgpu.net/docs/
- https://developer.nvidia.com/
- https://developer.nvidia.com/cuda
- https://docs.nvidia.com/cuda/cuda-programming-guide/
- https://learn.microsoft.com/en-us/dotnet/api/system.runtime.intrinsics

SIMD Data Parallelism

- The CPU uses this trick to parallelize simple operations

SIMD - Single Instruction, Multiple Data

- A single instruction (AVX, SSE, NEON) operates on multiple data items in a register simultaneously
- The CPU has wide registers, e.g. 256b (AVX)
- It can perform the same operation on several data items at once:

```
// SIMD (AVX)
a = [1,2,3,4]
b = [5,6,7,8]
c = a + b  // result: [6,8,10,12]
```

SIMD vs Scalar

Scalar operations are those where the processor handles one data element at a time. To multiply 8 consecutive numbers on an ordinary processor, we need 8 operations.

```
W[0]=X[0]*2
W[1]=X[1]*2
W[2]=X[2]*2
W[3]=X[3]*2
W[4]=X[4]*2
W[5]=X[5]*2
W[6]=X[6]*2
W[7]=X[7]*2
```

SIMD (vector) operations execute the same instruction on the whole data set at the same time, using the processor's wide registers (here 1 instruction and 8 operations).

X[0]*2	X[1]*2	X[2]*2	X[3]*2	X[4]*2	X[5]*2	X[6]*2	X[7]*2

A simple scalar operation

Suppose we have the following task. Two *float* arrays contain numbers. Multiply each number from the first array by the number at the same index in the second array. Store the result in a third array.

```
int n = 100_000_000;
float[] a = new float[n];
float[] b = new float[n];
float[] scalar = new float[n];

for (int i = 0; i < n; i++)
   scalar[i] = a[i] * b[i];
```

System.Runtime.Intrinsics

- Contains types used to create and pass the state of registers of various sizes and formats, intended for use with instruction set extensions.
- Instructions for manipulating these registers can be found in **System.Runtime.Intrinsics.X86** and **System.Runtime.Intrinsics.Arm**.
- Provides a set of static methods for creating, modifying and performing other operations on 64-, 128-, 256- and 512-bit vectors.

Source: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.intrinsics?view=net-10.0

SIMD version

```
static float[] MultiplySIMD(float[] a, float[] b)
{
    int n = a.Length;
    float[] result = new float[n];

    // Determine the number of elements that can be processed in parallel using SIMD
    int simdLength = Vector<float>.Count;
    int i = 0;

    // SIMD loop gets blocks of simdLength elements
    for (; i <= n - simdLength; i += simdLength)
    {
        Vector<float> va = new Vector<float>(a, i);
        Vector<float> vb = new Vector<float>(b, i);
        (va * vb).CopyTo(result, i);
    }

    // Tail (scalar)
    for (; i < n; i++)
    result[i] = a[i] * b[i];

    return result;
}
```

The remainder of the division by **simdLength** is processed in the classic (scalar) way

SIMD Parallel version

```
Parallel.For(0, processorCount, core =>
  {
      int chunkSize = n / processorCount;
      int start = core * chunkSize;
      int end = (core == processorCount - 1) ? n : start + chunkSize;

      int i = start;

      // SIMD part
      for (; i <= end - simdLength; i += simdLength)
      {
          var va = new Vector<float>(a, i);
          var vb = new Vector<float>(b, i);
          (va * vb).CopyTo(result, i);
      }

      // Rest calculation as scalar
      for (; i < end; i++)
          result[i] = a[i] * b[i];
  });
```

CPU != GPU

- A classic processor has a very simple work cycle:
  - Fetch (from computer memory)
  - Decode
  - Execute
- This is done sequentially, and on fast processors this is not a problem.
- Even if we split the task across processor cores, dividing the data into smaller chunks, we still perform the operations sequentially on each of them.

Concurrency on the GPU

- Thousands of cores executing SIMT (Single Instruction Multiple Threads).
- Huge memory bandwidth.
- High parallelism: hundreds of thousands of threads active at the same time.
- Ideal for computations of the type: for each element, perform the same operation (matrix operations).

Differences between SIMD and SIMT

SIMT — Single Instruction, Multiple Threads

- One instruction controls many threads, each of which has its own data and registers.
- The GPU runs thousands of threads in groups (warp = 32 threads in NVIDIA).
- All threads in a warp execute the same instruction, but on different data.
- If threads branch (e.g. if), the GPU executes them sequentially — this is called divergence and we should avoid it.

Concurrency on the GPU

How many cores do example models have

Model	CUDA cores	Tensor cores (5th gen)	RT cores (4th gen)
RTX 5090	21,760	680	170
RTX 5080	10,752	336	84
RTX 5070 Ti	8,960	280	70
RTX 5070	6,144	192	48
RTX 5060 Ti	4,608	144	36
RTX 5060	3,840	120	30
RTX 5050	2,560	80	20

WARP

How many threads can they handle? What is a WARP?

- In hardware, this is an integrated circuit of the graphics card with physical management units called *Warp Schedulers*. In one clock cycle they fetch and dispatch for execution an instruction only for a batch consisting of exactly 32 threads
- It works on the principle of SIMT

https://docs.nvidia.com/cuda/archive/12.8.0/blackwell-tuning-guide/index.html

How many threads at once?

- The number of active WARPs depends on the number of SMs (Streaming Multiprocessors)
- For example, in the RTX 50 series each SM handles 48 WARPs

Model	Number of SM blocks	Max number of warps at once	Total number of threads processed
RTX 5090	170	8,160	261,120
RTX 5080	84	4,032	129,024
RTX 5070 Ti	70	3,360	107,520
RTX 5070	48	2,304	73,728
RTX 5060 Ti	36	1,728	55,296
RTX 5060	30	1,440	46,080
RTX 5050	20	960	30,720

Concurrency on the GPU

We have 3 options

- **CUDA** via ManagedCUDA or ILGPU
- **OpenCL** via Cloo / OpenCL.Net
- Compute Shader (**HLSL**) via DirectX / DX12 / SharpDX / Vortice

CUDA

CUDA is NVIDIA's platform for hardware-accelerated computing, providing a software layer that allows applications to harness the computing power of graphics processing units (GPUs). Developers can write code in languages such as C++, Python and Fortran, or use GPU-accelerated libraries and frameworks, for example PyTorch. This flexibility allows GPU computing to be integrated into any layer of the technology stack, which translates into optimal functionality and performance.

Source: https://developer.nvidia.com/cuda

CUDA Toolkit + wrapper

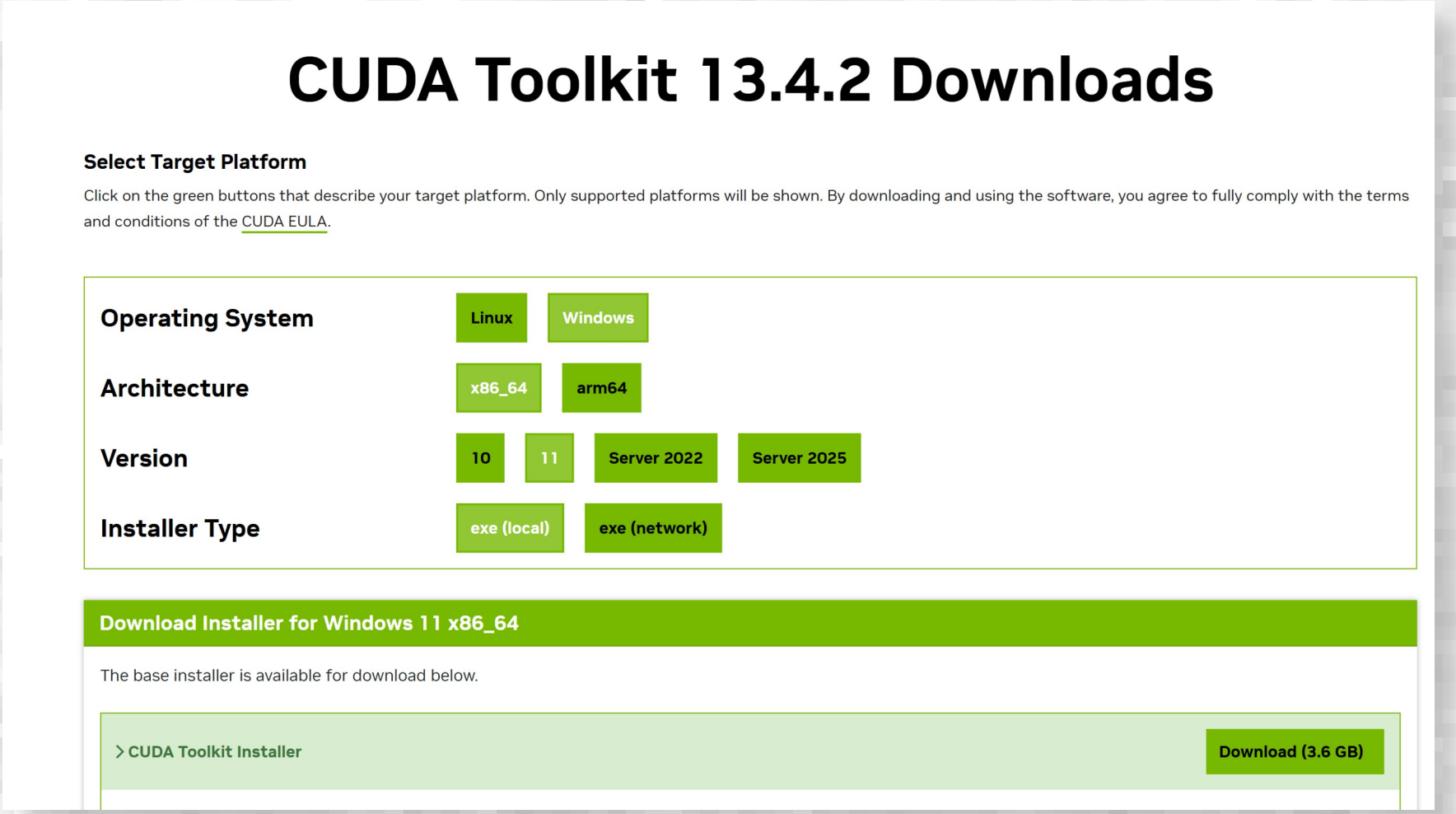
CUDA Toolkit – a key component of the CUDA platform – provides the compiler, libraries and development tools needed to build applications that use graphics processors.

ILGPU - a GPU JIT compiler for .NET: https://ilgpu.net/docs/

Source: https://developer.nvidia.com/cuda

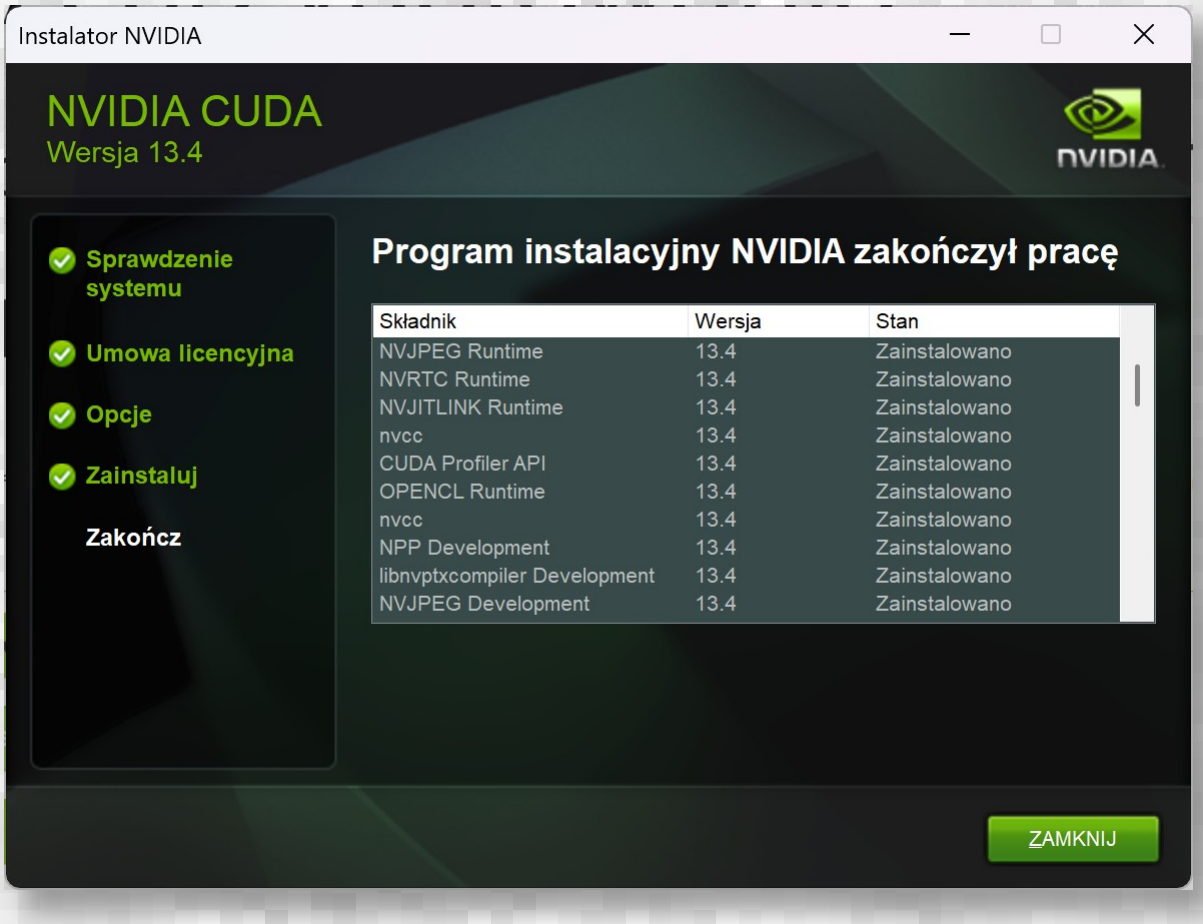
CUDA setup

We start by downloading and installing the CUDA Toolkit



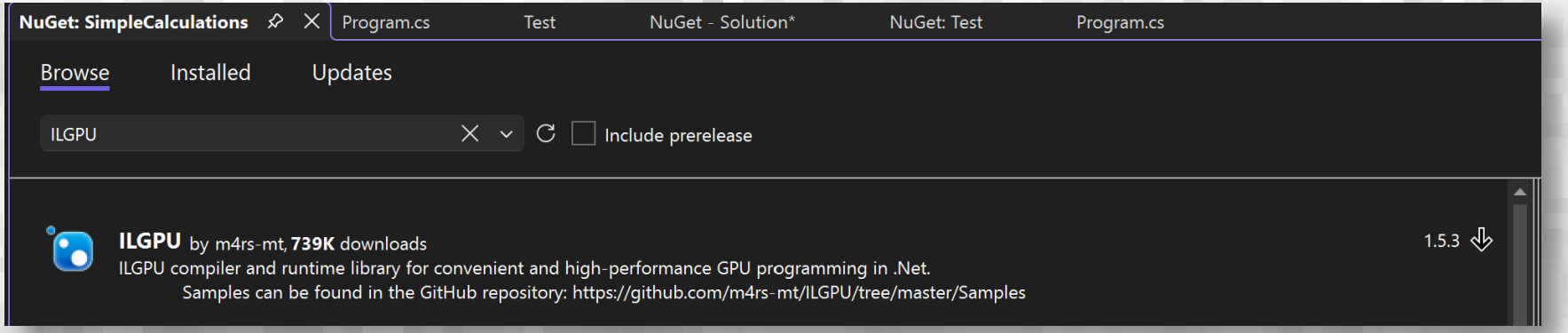
CUDA setup

In the end we should have a list of components



CUDA First program

- In a new project we add ILGPU via NuGet



We add simple kernel method code (i.e. what is run on the GPU)

```
static void MultiplyKernel(Index1D index, ArrayView<int> data)
  {
      data[index] *= 2;
  }
```


ConcurrentGPU/SimpleCalculations

CUDA Simple operations

- We create the context and accelerator objects

```
using var context = Context.CreateDefault();
using var accelerator = context.CreateCudaAccelerator(0);
```

- We initialize the variables and input data

```
int n = 10_000_000;
int[] dataCpu = Enumerable.Range(1, n).ToArray();
int[] dataGpu = Enumerable.Range(1, n).ToArray();
Stopwatch swCpu = new Stopwatch();
```

- Simple operations on the CPU (for comparison)

```
swCpu.Start();
for (int i = 0; i < n; i++)
{
    dataCpu[i] *= 2;
}
swCpu.Stop();
Console.WriteLine($"CPU time: {swCpu.ElapsedMilliseconds} ms");
```


ConcurrentGPU/SimpleCalculations

CUDA simple operations

- We prepare the buffer for the GPU

```
using MemoryBuffer1D<int, Stride1D.Dense> buffer =
accelerator.Allocate1D<int>(n);
```

- We pass the data to the GPU and prepare the kernel action

```
buffer.CopyFromCPU(dataGpu);
Action<Index1D, ArrayView<int>> kernel = accelerator
.LoadAutoGroupedStreamKernel<Index1D,ArrayView<int>>(MultiplyKernel);
```

- Executing the computation and synchronization

```
kernel(n, buffer.View);
accelerator.Synchronize();
```

- Copying back to the buffer

```
buffer.CopyToCPU(dataGpu);
```

CUDA result

Result for 10,000,000 elements:

```
CPU time: 10 ms

GPU kernel only: 2 ms

GPU time (including transfers): 164 ms
```

That is not what we expected

CUDA Heavier computations

- A more demanding method

```
static void HeavyKernel(Index1D index, ArrayView<float> data)
{
    float x = data[index];
    for (int i = 0; i < 500; i++)
        x = MathF.Sin(x) * MathF.Cos(x) + MathF.Sqrt(x + 1);
    data[index] = x;
}
```

- Additionally, a concurrent version on the CPU

```
Parallel.For(0, n, i =>
    {
        float x = dataCpu2[i];
        for (int j = 0; j < 500; j++)
            x = MathF.Sin(x) * MathF.Cos(x) + MathF.Sqrt(x + 1);
    dataCpu2[i] = x;
    });
```


ConcurrentGPU/SimpleCalculations

CUDA result

Result for 10,000,000 elements:

```
GPU kernel only: 10 ms
CPU time: 136545 ms
CPU time with multi threading: 15072 ms on 24 cores
GPU time (including transfers): 123 ms
```

Now the difference is visible

Kernel

- A kernel is the basic program executed on the graphics card. The pattern is very similar to *Parallel.For* i.e.:
- We have a start and end index
- We have access to some collection of elements
- We execute a piece of code for a specific element at a given index in the collection.
- The execution order is not deterministic.

Source: https://developer.nvidia.com/cuda

Memory

- When computers were still 8-bit, the processor could be slower than the memory.
- Now memory is the bottleneck of computation. This happens primarily because of physical limitations. The farther the memory cells are from the processor, the more time a signal needs to travel to/from memory.
- That is why fast memory is the memory built into the processor, e.g. registers.
- That is why processors use various tricks, e.g. *prefetching* (fetching data into the processor cache)

Memory

Bandwidth:

- DDR5-6400 is 51.2 GB/s per channel, i.e. 102.4 GB/s in dual-channel mode
- RTX 5080 GDDR7 is 960GB/s
- Nvidia H100 with HBM3 is already around 3000GB/s

The advantage in favor of graphics cards is as much as several dozen times.

Context

- All classes and functions of the ILGPU library use an instance of *ILGPU.Context*.
- The context's main task is to act as an interface to the ILGPU compiler.
- It also stores some global state.
- By default we create it with

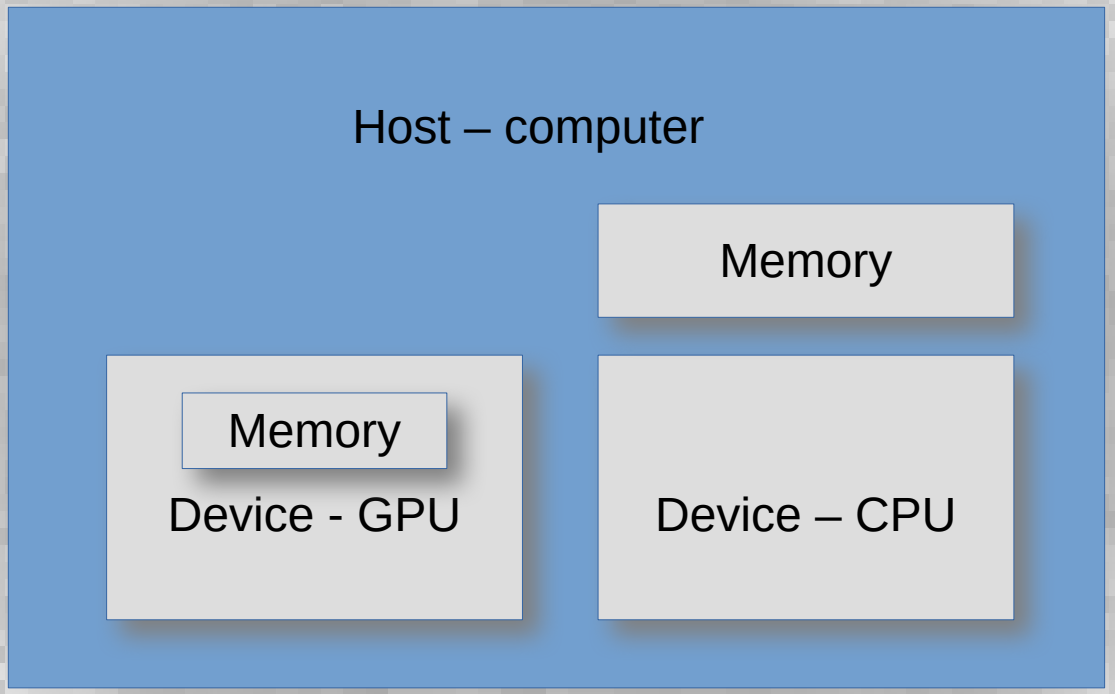
```
using Context context = Context.CreateDefault();
```

- Context is *IDisposable* so we should call Dispose or place it in a using block

Accelerators

- In ILGPU, an accelerator represents a hardware or software graphics processor (GPU).
- Every ILGPU program requires at least one accelerator.
- The following accelerator types are currently available:
  - CPU, `using CPUAccelerator accelerator = context.CreateCPUAccelerator(0);`
  - CUDA, `using CudaAccelerator accelerator = context.CreateCudaAccelerator(0);`
  - OpenCL, `using CLAccelerator accelerator = context.CreateCLAccelerator(0);`

Buffers



In *ILGPU* memory is described in terms of arrays; even when we want to pass a single value, we pass a one-element array.

MemoryBuffer1D<T>

- **MemoryBuffer** is a host-side copy of memory allocated on the device. In essence, it is simply a pointer to a memory area allocated on the device.
- This object is always obtained through the accelerator (*Accelerator*).
- Requires the namespace:

```
using ILGPU.Runtime;
```

- Example of how to create it:

```
using MemoryBuffer1D<int, Stride1D.Dense>
bufferOnDevice = accelerator.Allocate1D<int>(n);
or
MemoryBuffer1D<int, Stride1D.Dense> deviceData =
accelerator.Allocate1D(new int[] { 0, 1, 2, 3, 4, 5,
6, 7, 8, 9 });
```

Copy From/To CPU

- After allocating a MemoryBuffer object, we load the data:

```
bufferOnDevice.CopyFromCPU(dataGpu);
```

- Or we retrieve it:

```
bufferOnDevice.CopyToCPU(dataGpu);
 or
int[] hostOutput = bufferOnDevice.GetAsArray1D();
```

ArrayView<T>

- *ArrayView* is a copy of memory located on the device (GPU) side, which was allocated from the host (CPU) level.
- It is an element of the *MemoryBuffer*API, which kernels and the GPU interact with directly.
- This object is always obtained through *MemoryBuffer*.
- Requires: `using ILGPU.Runtime;`
- Basic way to create it: `ArrayView1D<int, Stride1D.Dense> ints = bufferOnDevice.View;`
- Inside a kernel, the *ArrayView* object behaves exactly as you would expect from an ordinary array.

Memory examples

```
static void Main()
{
    Context context = Context.CreateDefault();
    Accelerator accelerator = context.CreateCPUAccelerator(0);
    double[] doubles = new double[1000];
    MemoryBuffer1D<double, Stride1D.Dense> doublesOnDevice =
accelerator.Allocate1D(doubles);
    for (int i = 0; i < doubles.Length; i++)
          { doubles[i] = i * Math.PI; }
    doublesOnDevice.CopyFromCPU(doubles);
    // do some operations on device
    doublesOnDevice.CopyToCPU(doubles);
    double[] doubles2 = new double[doublesOnDevice.Length];
    doublesOnDevice.CopyFromCPU(doubles2);
 //As above 2, this completely allocates a new double[] on the host.
This is slow.
    double[] doubles3 = doublesOnDevice.GetAsArray1D();
    ArrayView1D<double, Stride1D.Dense> doublesArrayView =
doublesOnDevice.View;
    doublesOnDevice.Dispose();
    accelerator.Dispose();
    context.Dispose();
}
```

ConcurrentGPU/MemoryExample

Kernel

```
// Initialize ILGPU.
Context context = Context.CreateDefault();
Accelerator accelerator = context.GetPreferredDevice(preferCPU: false)
                                 .CreateAccelerator(context);
// Load the data.
MemoryBuffer1D<int, Stride1D.Dense> deviceData = accelerator.Allocate1D(new
int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 });
MemoryBuffer1D<int, Stride1D.Dense> deviceOutput =
accelerator.Allocate1D<int>(10_000);
// load / precompile the kernel
Action<Index1D, ArrayView<int>, ArrayView<int>> loadedKernel =
       accelerator.LoadAutoGroupedStreamKernel<Index1D,
       ArrayView<int>, ArrayView<int>>(KernelMethod);
// finish compiling and tell the accelerator to start computing the kernel
loadedKernel((int)deviceOutput.Length, deviceData.View, deviceOutput.View);
// wait for the accelerator to be finished
accelerator.Synchronize();
// moved output data from the GPU to the CPU
int[] hostOutput = deviceOutput.GetAsArray1D();
for (int i = 0; i < 50; i++) {Console.Write($"{hostOutput[i]} ");}

accelerator.Dispose();
context.Dispose();
```


ConcurrentGPU/KernelExample

Kernel – main method

```
static void KernelMethod(Index1D i, ArrayView<int> data,
ArrayView<int> output)
  {
      output[i] = data[i % data.Length];
  }
```

- You can only allocate and pass non-nullable value types of fixed size
- No classes
- No references
- No dynamically sized structures
- The first parameter of a kernel must be its index.
- A kernel always iterates over a defined range, which can be 1, 2 or 3 dimensions
- The function can contain any logic, but let us avoid conditional branching

ConcurrentGPU/KernelExample

SIMD vs CUDA

```
static void MultiplySIMT(Index1D index, ArrayView<float> a, ArrayView<float> b,
ArrayView<float> result)
     {
         result[index] = a[index] * b[index];
     }
static float[] MultiplyGPU(float[] a, float[] b, Accelerator accelerator)
   {
       int n = a.Length;
       using var da = accelerator.Allocate1D<float>(n);
       using var db = accelerator.Allocate1D<float>(n);
       using var dr = accelerator.Allocate1D<float>(n);
       da.CopyFromCPU(a);
       db.CopyFromCPU(b);
       var kernel = accelerator.LoadAutoGroupedStreamKernel<Index1D, ArrayView<float>,
ArrayView<float>, ArrayView<float>>(MultiplySIMT);
       kernel(n, da.View, db.View, dr.View);
       accelerator.Synchronize();
       float[] result = new float[n];
       dr.CopyToCPU(result);
       return result;
   }
```

```
static void MultiplySIMT(Index1D index,
ArrayView<float> a, ArrayView<float> b,
ArrayView<float> result)
        {
            result[index] = a[index] * b[index];
        }
```

Results

10 trials, results in ms. As you can see, the GPU did not show its full power

Trial	Scalar	SIMD	SIMD P	GPU
1	193	157	109	409
2	179	146	123	353
3	182	156	158	388
4	303	264	61	414
5	186	184	59	339
6	187	159	56	368
7	270	197	59	416
8	181	155	60	355
9	253	184	56	444
10	218	196	115	411
average	**215.2**	**179.8**	**85.6**	**389.7**

CUDA vs SIMD vs Scalar

round two

Now let us run a similar test for somewhat more demanding computations

```
static void HeavyScalar(float[] input, float[] output)
{
    for (int i = 0; i < input.Length; i++)
    {
        float x = input[i];
        for (int j = 0; j < iterateTo; j++)
            x = x * 1.0001f + 0.0001f;
        output[i] = x;
    }
}
```

AVX2 – a new contender

- In theory, the compiler selects the most optimal solution for the given computations by itself, but we can also decide ourselves which processor instructions to use.
- For this we will use the *AVX2* class with its optimized methods
- The problem is that the methods *Avx.LoadVector256* and *Avx.Store* only accept a pointer (*float**) to data in memory. Therefore we have to use ***unsafe***

AVX2 – a new contender

```
 for (; i <= input.Length - simdWidth; i += simdWidth)
 {
     unsafe //to compile unsafe code, you need to enable "Allow unsafe code"
in project settings
     {
         fixed (float* pInput = &input[i])
         fixed (float* pOutput = &output[i])
         {
             // write 8 floats to register
             Vector256<float> v = Avx.LoadVector256(pInput);
             for (int j = 0; j < iterateTo; j++)
             {
                 v = Avx.Add(Avx.Multiply(v, mul), add);
             }
             Avx.Store(pOutput, v);
         }
     }
 }
```

AVX2 FMA – a new contender

- Alternatively, instead of the two methods Add and Multiply,

```
for (int j = 0; j < iterateTo; j++)
{
    v = Avx.Add(Avx.Multiply(v, mul), add);
}
```

- we can use a single dedicated *MultiplyAdd*

```
for (int j = 0; j < iterateTo; j++)
{
    v = Fma.MultiplyAdd(v, mul, add);
}
```

Comparison

Ten trials, results in ms

Trial	Scalar	SIMD	AVX2	FMA	FMA P	GPU
1	32434	8720	8458	5777	335	230
2	31744	9189	9506	6181	365	207
3	31260	8901	9166	6162	365	212
4	34196	9376	9375	7015	386	186
5	31271	9178	9236	6221	304	120
6	30896	9428	8960	5650	386	149
7	30142	9161	9364	6284	325	124
8	34256	9537	9756	7193	462	184
9	32162	9361	9295	6188	308	126
10	34600	9320	9220	6294	395	210
average	**32296.1**	**9217.1**	**9233.6**	**6296.5**	**363.1**	**174.8**

If the difference between GPU and FMA P for 1024 iterations seems small, here is an example for 256*1024

FMA P: 168039 ms

GPU kernel only: 148 ms

GPU total (incl. transfers): 378 ms

Short SIMD summary

Instructions	What they do
FMA — Fused Multiply-Add (Fma.MultiplyAdd)	Computes v = v*m + a
Avx.Multiply / Avx.Add / Avx.Subtract / Avx.Divide	Performs basic SIMD operations on 8 floats at once
Avx.Sqrt	Vector square root
Vector<float>.SquareRoot	as above, but less efficient in longer loops
Avx.And / Avx.Or / Avx.Xor / Avx.AndNot	Bitwise SIMD operations
Avx.Compare / Avx.CompareGreaterThan / Avx.CompareLessThan	Compare 8 floats at once and return a bit mask
Avx.Blend / Avx.BlendVariable	Combine two vectors based on a mask
Avx.Permute / Avx.Shuffle	Rearrange the elements of a vector
Avx.HorizontalAdd / HorizontalSubtract	Add or subtract vector elements across the vector.
Avx.BroadcastScalarToVector256	Replicates one value across the whole vector.

Short SIMD summary

Instructions	What they do
Avx.LoadVector256 / Avx.Store	Load and store 8 floats at once.
Avx.MaskLoad / Avx.MaskStore	Load/store only selected elements of the vector.
Avx2.GatherVector256	Loads elements from non-contiguous addresses (vector indexing).
Avx2.PermuteVar8x32	Advanced element permutations
Avx2.ShiftLeft / ShiftRight	SIMD bit shifts.
Avx2.MultiplyLow / MultiplyHigh	Integer multiplication keeping the low/high bits.
Avx2.AddSaturate / SubtractSaturate	Addition/subtraction with saturation (no overflow).
Avx2.CompareEqual / CompareGreaterThan	SIMD integer comparisons.
Avx2.Blend / BlendVariable	Combining vectors based on a mask.
Avx2.Extract / Insert	Extract or insert elements of a vector.
Avx2.Pack / Unpack	Pack or unpack data (e.g. 16-bit → 8-bit).

Matrix multiplication

- Single thread

```
swCpu.Start();
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
    {
        float sum = 0;
        for (int k = 0; k < N; k++)
            sum += A[i, k] * B[k, j];
        Ccpu[i, j] = sum;
    }
swCpu.Stop();
```


ConcurrentGPU/MatrixMultiplication

Matrix multiplication

- Concurrent CPU version

```
swCpuParallel.Start();
 Parallel.For(0, N, i =>
 {
     for (int j = 0; j < N; j++)
     {
         float sum = 0;
         for (int k = 0; k < N; k++)
             sum += A[i, k] * B[k, j];
         CcpuParallel[i, j] = sum;
     }
 });
 swCpuParallel.Stop();
```

Matrix multiplication

- GPU

```
using var dA = accelerator.Allocate2DDenseX<float>(new Index2D(N, N));
using var dB = accelerator.Allocate2DDenseX<float>(new Index2D(N, N));
using var dC = accelerator.Allocate2DDenseX<float>(new Index2D(N, N));
swGpuTotal.Start();
dA.CopyFromCPU(A);
dB.CopyFromCPU(B);
var kernel = accelerator.LoadAutoGroupedStreamKernel<
    Index2D,
    ArrayView2D<float, Stride2D.DenseX>,
    ArrayView2D<float, Stride2D.DenseX>,
    ArrayView2D<float, Stride2D.DenseX>>(MatrixMulKernel);
swGpuKernel.Start();
kernel(new Index2D(N, N), dA.View, dB.View, dC.View);
accelerator.Synchronize();
swGpuKernel.Stop();
dC.CopyToCPU(Cgpu);
swGpuTotal.Stop();
```

Matrix multiplication

- GPU

```
static void MatrixMulKernel(
    Index2D index,
    ArrayView2D<float, Stride2D.DenseX> A,
    ArrayView2D<float, Stride2D.DenseX> B,
    ArrayView2D<float, Stride2D.DenseX> C)
{
    int row = index.X;
    int col = index.Y;

    float sum = 0;
    // Use Extent.X instead of Width
    for (int k = 0; k < A.Extent.X; k++)
        sum += A[row, k] * B[k, col];
    C[row, col] = sum;
}
```

Matrix multiplication

- GPU Tiled (discussed later)

```
dA.CopyFromCPU(A);
dB.CopyFromCPU(B);

var kernelTiled = accelerator.LoadStreamKernel<
     ArrayView2D<float, Stride2D.DenseX>,
     ArrayView2D<float, Stride2D.DenseX>,
     ArrayView2D<float, Stride2D.DenseX>,
     int>(MatrixMulTiled);

const int TILE = 16;
var blockSize = new Index2D(TILE, TILE);
var gridSize = new Index2D((N + TILE - 1) / TILE,(N + TILE - 1) / TILE);
var config = new KernelConfig(gridSize, blockSize);
kernelTiled(config, dA.View, dB.View, dC.View, N);
accelerator.Synchronize();
dC.CopyToCPU(CgpuTile);
```

Matrix multiplication

- GPU Tiled

```
static void MatrixMulTiled(ArrayView2D<float, Stride2D.DenseX> A,
 ArrayView2D<float, Stride2D.DenseX> B, ArrayView2D<float, Stride2D.DenseX> C, int N)
 {
     const int TILE = 16;
     var sharedA = SharedMemory.Allocate<float>(TILE * TILE);
     var sharedB = SharedMemory.Allocate<float>(TILE * TILE);
     int tx = Group.IdxX;
     int ty = Group.IdxY;
     int blockX = Grid.IdxX;
     int blockY = Grid.IdxY;
     int row = blockY * TILE + ty;
     int col = blockX * TILE + tx;
     float sum = 0.0f;
     int numTiles = (N + TILE - 1) / TILE;
     for … ---→ continuation on next slide
```

Matrix multiplication

- GPU Tiled

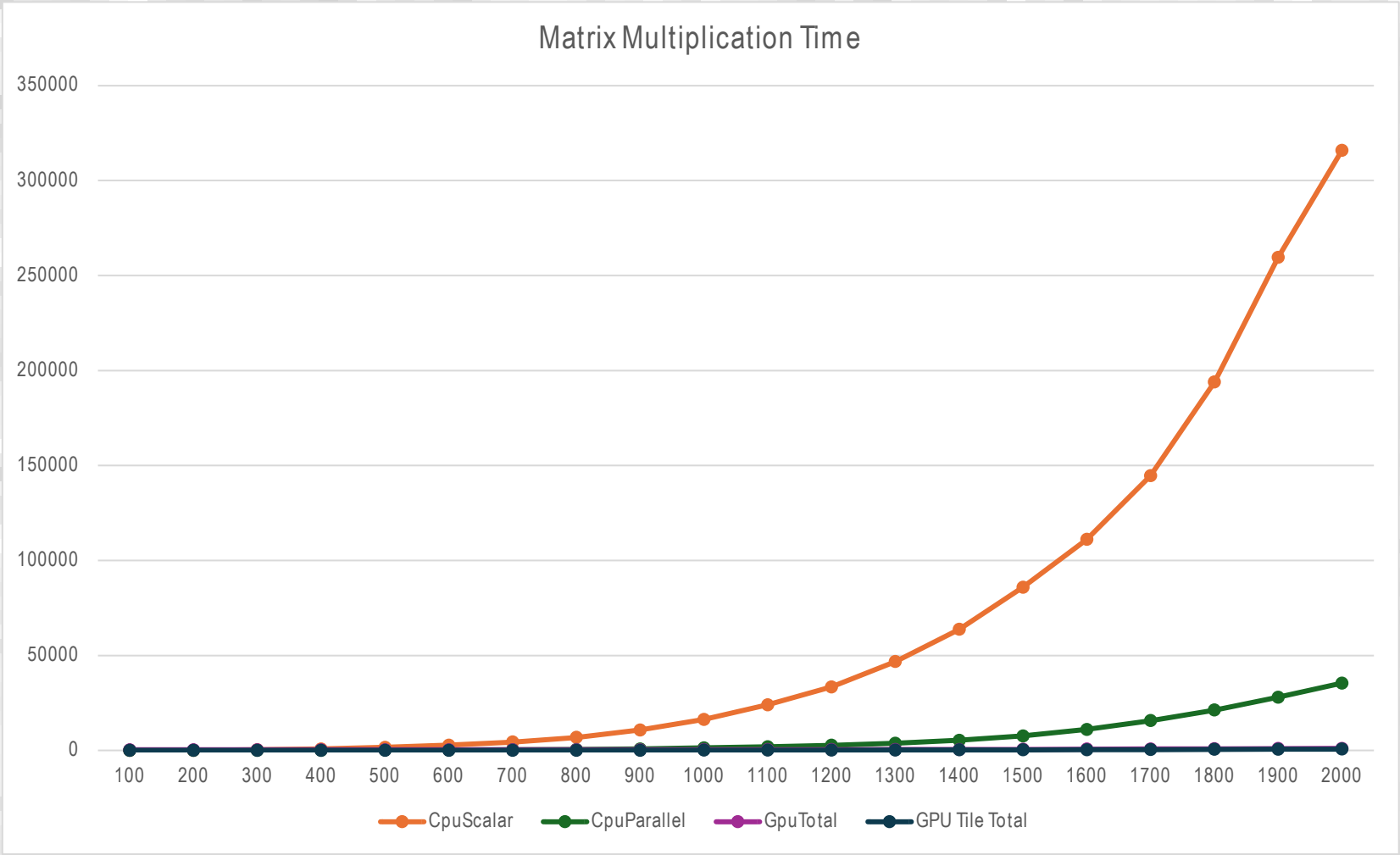
```
--->…
   for (int t = 0; t < numTiles; t++)
     {
         int aCol = t * TILE + tx;
         int bRow = t * TILE + ty;
         // guarded loads
         sharedA[ty * TILE + tx] = (row < N && aCol < N) ? A[row, aCol] : 0.0f;
         sharedB[ty * TILE + tx] = (bRow < N && col < N) ? B[bRow, col] : 0.0f;
         Group.Barrier();
         for (int k = 0; k < TILE; k++)
         {
             sum += sharedA[ty * TILE + k] * sharedB[k * TILE + tx];
         }
         Group.Barrier();
     }
     if (row < N && col < N)
         C[row, col] = sum;
 }
```

Matrix multiplication

Size	CpuScalar	CpuParallel	GpuKernel	GpuTotal	GpuTile	GpuTileTotal
100	4	4	5	319	3	41
200	80	11	5	321	3	43
300	330	31	5	324	3	45
400	779	80	6	327	4	48
500	1629	156	6	334	4	51
600	2769	227	7	338	7	59
700	4408	344	8	345	7	65
800	6874	507	9	353	8	73
900	10735	761	16	370	9	85
1000	16336	1324	18	393	10	108
1100	24037	1894	21	416	12	127
1200	33423	2690	28	441	14	161
1300	46739	3772	32	472	16	202
1400	63708	5339	36	514	19	253
1500	85899	7660	42	579	22	296
1600	111105	11015	50	678	27	355
1700	144599	15702	60	753	31	429
1800	193899	21228	71	854	37	531
1900	259560	28028	83	978	45	640
2000	315932	35441	98	1065	60	724

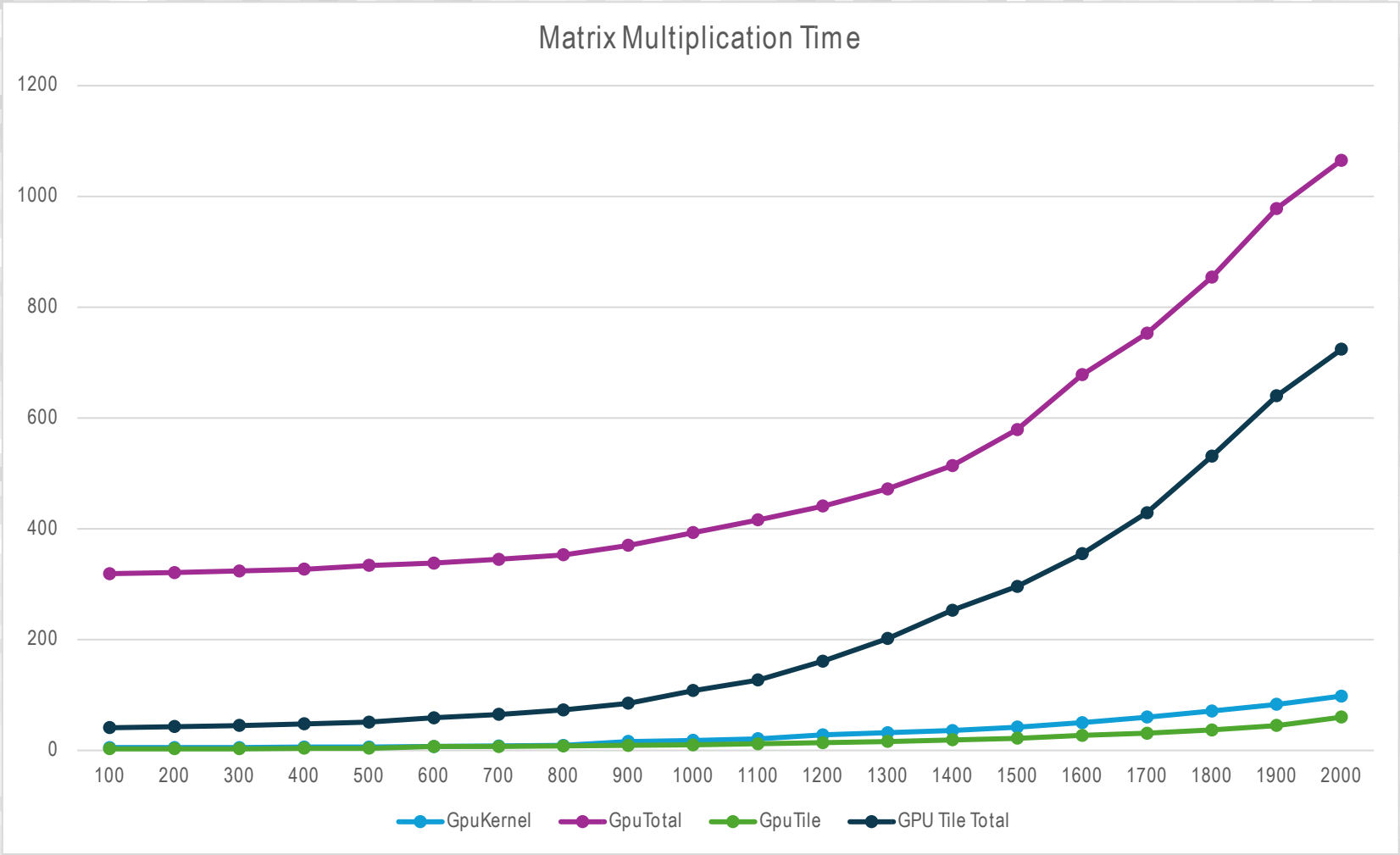
Matrix multiplication

- Results: CPU single, CPUP, GPU and GPU Tile



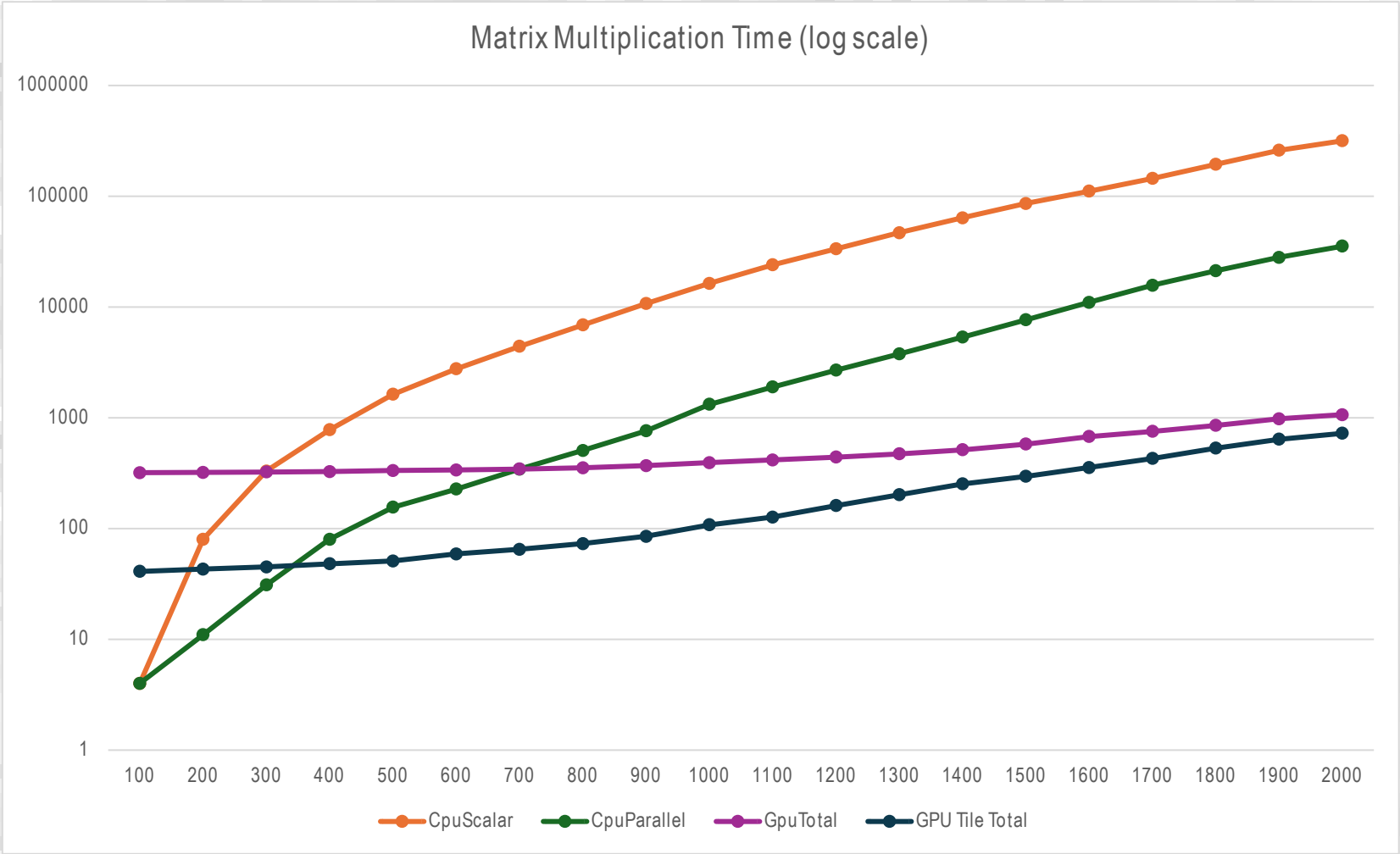
Matrix multiplication

- GPU only vs GPU Tile



Matrix multiplication

- Results: CPU single, CPUP, GPU and GPU Tile



Structures

- On the GPU we cannot always use data in the same way as on the CPU when using C#. That is why ILGPU imposes certain restrictions related to lower-level coding.
- For example, we cannot use structs inside classes.

ConcurrentGPU/NbodyExample NBodyWPFExample

Struct of Arrays vs Array of Structs

- Suppose we have a Car struct with basic parameters

```
public struct Car
{
    public float PositionX;
    public float PositionY;
    public float Velocity;
    public float Fuel;
    public float Temperature;
    public float EngineWear;
    public float TireWear;
    public float Weight;
}
```

- After creating and initializing an array of these structs, we pass it to the GPU

```
using MemoryBuffer1D<Car, Stride1D.Dense> deviceCars =
accelerator.Allocate1D(hostCars);
```


ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- Then we run the computation on the GPU

```
  var kernel = accelerator.LoadAutoGroupedStreamKernel<Index1D,
ArrayView<Car>>(UpdateCarsKernel);
```

- Where we perform some simple position calculation

```
static void UpdateCarsKernel(Index1D index,ArrayView<Car> cars)
{
    Car car = cars[index];
    for (int i = 0; i < Iterations; i++)
    {
        car.PositionX += car.Velocity;
        car.PositionY -= car.Velocity;
    }
    cars[index] = car;
}
```

- After the computation, we return the data to the host using *CopyToCPU*

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- After 10 trials with 1 million cars and 100 iterations, the results are as follows (in ms):

Trial	InitializationTime	PreparationTime	KernelTime	ReturningTime	Total
1	1424	790	16	563	2793
2	1533	584	12	574	2703
3	1532	725	12	563	2832
4	4535	1987	185	665	7372
5	3976	2182	185	790	7133
6	2033	948	12	709	3702
7	2231	756	14	905	3906
8	2501	824	12	589	3926
9	1936	750	12	670	3368
10	4758	3092	185	736	8771
Avg.	**2645.9**	**1263.8**	**64.5**	**676.4**	**4650.6**

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- For comparison, Struct of Arrays

```
float[] hostPositionX = new float[CarCount];
float[] hostPositionY = new float[CarCount];
float[] hostVelocity = new float[CarCount];
float[] hostFuel = new float[CarCount];
float[] hostTemperature = new float[CarCount];
float[] hostEngineWear = new float[CarCount];
float[] hostTireWear = new float[CarCount];
float[] hostWeight = new float[CarCount];
```

- We allocate each array on the GPU separately

```
using var dPositionX = accelerator.Allocate1D(hostPositionX);
using var dPositionY = accelerator.Allocate1D(hostPositionY);
using var dVelocity = accelerator.Allocate1D(hostVelocity);
using var dFuel = accelerator.Allocate1D(hostFuel);
using var dTemperature = accelerator.Allocate1D(hostTemperature);
using var dEngineWear = accelerator.Allocate1D(hostEngineWear);
using var dTireWear = accelerator.Allocate1D(hostTireWear);
using var dWeight = accelerator.Allocate1D(hostWeight);
```


ConcurrentGPU/CarsSoA

Struct of Arrays vs Array of Structs

- We pass to the GPU only what we need

```
var kernel = accelerator.LoadAutoGroupedStreamKernel<Index1D,
ArrayView<float>, ArrayView<float>,ArrayView<float>>
(UpdateCarsKernel);
```

- The kernel method does the same computations but uses 3 separate arrays

```
static void UpdateCarsKernel(Index1D index, ArrayView<float> positionX,
ArrayView<float> positionY, ArrayView<float> velocity)
{
    float x = positionX[index];
    float y = positionY[index];
    float v = velocity[index];
    for (int i = 0; i < Iterations; i++)
    {
        x += v;
        y -= v;
    }
    positionX[index] = x;
    positionY[index] = y;
}
```


ConcurrentGPU/CarsSoA

Struct of Arrays vs Array of Structs

- After 10 trials with 1 million cars and 100 iterations, the results are as follows (in ms):

Trial	InitializationTime	PreparationTime	KernelTime	ReturningTime	Total
1	1546	866	6	215	2633
2	1772	635	4	170	2581
3	1773	1007	3	144	2927
4	1848	977	54	151	3030
5	1934	1285	54	292	3565
6	2202	1941	55	463	4661
7	2762	842	54	199	3857
8	3941	817	3	171	4932
9	1910	684	54	152	2800
10	1932	672	43	183	2830
Total	**2162**	**972.6**	**33**	**214**	**3381.6**

Struct of Arrays vs Array of Structs

- Let us try the same with a slimmed-down version of Car

```
public struct Car
{
    public float PositionX;
    public float PositionY;
    public float Velocity;

}
```

- Some of the data is not needed for the computation

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- After 10 trials with 1 million cars and 100 iterations, slimmed-down AoS version (in ms):

Trial	InitializationTime	PreparationTime	KernelTime	ReturningTime	Total
1	1030	444	7	212	1693
2	1085	207	9	202	1503
3	1064	210	4	216	1494
4	1157	235	4	270	1666
5	1527	626	4	317	2474
6	1645	276	66	225	2212
7	1125	259	4	223	1611
8	1214	266	49	263	1792
9	1594	320	13	220	2147
10	1174	280	69	216	1739
Avg.	**1261.5**	**312.3**	**22.9**	**236.4**	**1833.1**

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- In the SoA version, for comparison, we drop the extra arrays:

```
float[] hostPositionX = new float[CarCount];
float[] hostPositionY = new float[CarCount];
float[] hostVelocity = new float[CarCount];

//float[] hostFuel = new float[CarCount];
//float[] hostTemperature = new float[CarCount];
//float[] hostEngineWear = new float[CarCount];
//float[] hostTireWear = new float[CarCount];
//float[] hostWeight = new float[CarCount];
```

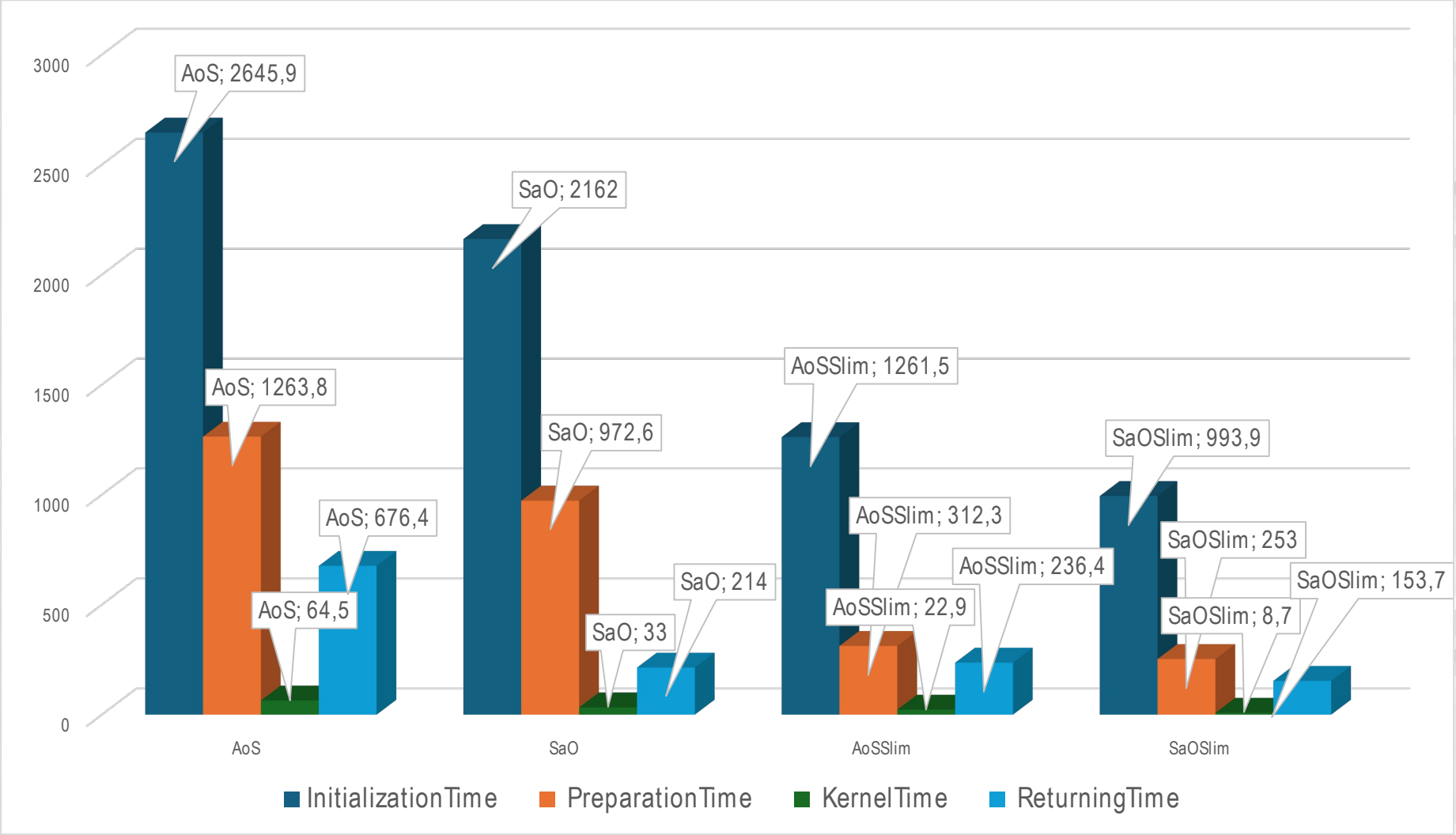
- *velocity* does not have to be passed back if we do not change it.

Struct of Arrays vs Array of Structs

- 10 trials, 1 million cars, 100 iterations (slimmed-down version), results in ms.

Trial	InitializationTime	PreparationTime	KernelTime	ReturningTime	Total
1	883	408	6	150	1447
2	885	210	4	141	1240
3	885	216	3	148	1252
4	910	211	4	139	1264
5	918	219	3	140	1280
6	1052	252	3	175	1482
7	1067	210	4	150	1431
8	1303	301	54	211	1869
9	1113	219	3	137	1472
10	923	284	3	146	1356
Avg.	**993.9**	**253**	**8.7**	**153.7**	**1409.3**

Struct of Arrays vs Array of Structs



Nbody – Another example

```
public struct Particle
 {
     public Vec3 position;
     public Vec3 velocity;
     public Vec3 acceleration;

     public Particle(Vec3 position)
     {
         this.position = position;
         velocity = new Vec3();
         acceleration = new Vec3();
     }
     private void updateAcceleration(ParticleSystem d, int ID)
     { ... }
     private void updatePosition()
     {
         position = position + velocity + acceleration * 0.5f;
     }
     private void updateVelocity()
     {
         velocity = velocity + acceleration;
     }
     public void update(ParticleSystem particles, int ID)
     { ... }
 }
```

A struct representing a particle, along with methods used to update its position, acceleration and velocity

ConcurrentGPU/NbodyExample NBodyWPFExample

Nbody – Another example

Separate structures

```
public MemoryBuffer1D<Vec3, Stride1D.Dense> positions;
public MemoryBuffer1D<Vec3, Stride1D.Dense> velocities;
public MemoryBuffer1D<Vec3, Stride1D.Dense> accelerations;

public ParticleSystemStructOfArrays(ArrayView1D<Vec3,
Stride1D.Dense> positions, ArrayView1D<Vec3, Stride1D.Dense>
velocities, ArrayView1D<Vec3, Stride1D.Dense> accelerations, int
width, int height)
    {
        this.positions = positions;
        this.velocities = velocities;
        this.accelerations = accelerations;
        gc = 0.001f;
        centerPos = new Vec3(0.5f * width, 0.5f * height, 0);
        centerMass = (float)positions.Length;
    }
```

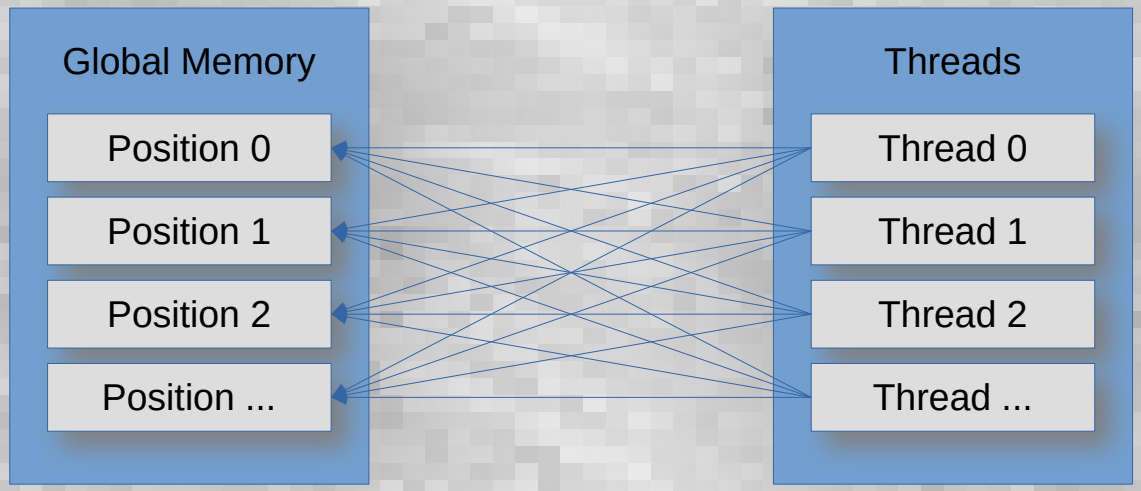

ConcurrentGPU/NbodyExample NBodyWPFExample2

Nbody – Tiled

- In the N-Body algorithm, each particle must read the positions of all the other particles. For 32,768 particles:

$$32\,768 \times 32\,768 \approx 1 \text{ billion interactions}$$

- Without tiling, each thread repeatedly reads data from the GPU global memory:

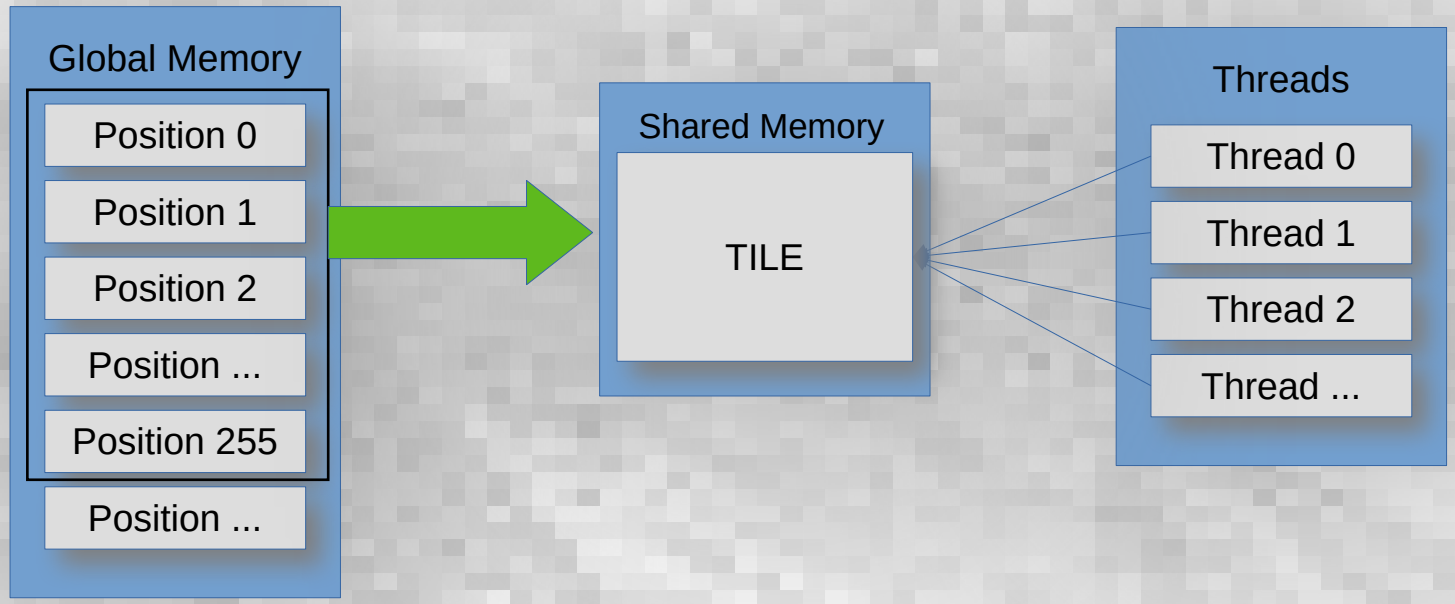


- Global memory has high bandwidth, but also high latency.
- The same data is read many times by different threads.

ConcurrentGPU/NbodyExample NBodyWPFExample3

Nbody – Tiled

A group of threads loads a chunk of data into fast shared memory (Shared Memory).



Load, synchronize (Group.Barrier()), perform computations using shared memory, load the next chunk.

ConcurrentGPU/NbodyExample NBodyWPFExample3

Nbody – Tiled

An additional modification is splitting the computation into two separate kernels:

`ParticleKernelTiled`
`RenderKernel`

Key elements:

```
var sharedPositions = SharedMemory.Allocate<Vec3>(TILE_SIZE);
...
if (sourceIdx < particleCount)
     sharedPositions[localIdx] = p.positions[sourceIdx];
...
Group.Barrier();

for (int j = 0; j < tileCount; j++)
  {
// .. calculate positions
  }
Group.Barrier();
```


ConcurrentGPU/NbodyExample NBodyWPFExample3

Nbody – Summary

V1 averages about 20 fps

V2 about 52 fps

V3 about 100 fps

The End

Thank you for your attention