Programowanie Współbieżne

C#

Akceleracja sprzętowa

Materiały

- 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 Równoległość danych

CPU stosuje ten trik by zrównoleglić proste operacje

SIMD - Single Instruction, Multiple Data

- Jedna instrukcja (AVX, SSE, NEON) działa jednocześnie na wielu danych w rejestrze
- CPU ma szerokie rejestry np. 256b (AVX)
- Może wykonać tę samą operację na kilku danych jednocześnie:

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

SIMD vs Scalar

Operacje skalarne to takie, w których procesor przetwarza jeden element danych na raz. Więc jeżeli chcemy wymnożyć kolejne 8 liczb na zwykłym procesorze musimy wykonać 8 operacji.

```
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
```

Operacje **SIMD** (wektorowe) wykonują tę samą instrukcję na całym zestawie danych w tym samym czasie. Wykorzystując szerokie rejestry procesora (tu 1 instrukcja i 8 operacji).

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

Prosta operacja skalarna

Załóżmy, że mamy takie zadanie. W dwóch tablicach *float* są liczby. Należy każdą liczbę z pierwszej tablicy przemnożyć przez liczbę o tym samym indeksie z drugiej tablicy. Wynik zapisać do trzeciej tablicy.

```
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

- Zawiera typy służące do tworzenia i przekazywania stanów rejestrów o różnych rozmiarach i formatach, przeznaczone do użytku z rozszerzeniami zestawu instrukcji.
- Instrukcje dotyczące manipulowania tymi rejestrami można znaleźć w **System.Runtime.Intrinsics.X86** i **System.Runtime.Intrinsics.Arm**.
- Udostępnia zestaw metod statycznych służących do tworzenia, modyfikowania i wykonywania innych operacji na wektorach 64,128, 256 i 512-bitowych.

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

Wersja SIMD

```
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;
}
```

Reszta z dzielenia przez `simdLength` jest realizowana w klasyczny (skalarny) sposób

Wersja SIMD Parallel

```
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

- Klasyczny procesor ma bardzo prosty cykl pracy:
  - Pobierz (z pamięci komputera)
  - Zdekoduj
  - Wykonaj
- Jest to wykonywane sekwencyjnie i na szybkich procesorach nie ma z tym problemu.
- Nawet jeżeli rozbijemy zadanie na rdzenie procesora, dzieląc dane na mniejsze kawałki, to nadal wykonujemy operacje sekwencyjnie na każdym z nich.

Współbieżność na GPU

- Tysiące rdzeni wykonujących SIMT (Single Instruction Multiple Threads).
- Ogromna przepustowość pamięci.
- Wysoka równoległość: setki tysięcy wątków aktywnych jednocześnie.
- Idealne do obliczeń typu: dla każdego elementu wykonaj tę samą operację. (operacje na macierzach)

Różnice między SIMD i SIMT

SIMT — Single Instruction, Multiple Threads

- Jedna instrukcja steruje wieloma wątkami, z których każdy ma własne dane i rejestry.
- GPU uruchamia tysiące wątków w grupach (warp = 32 wątki w NVIDIA).
- Wszystkie wątki w warpie wykonują tę samą instrukcję, ale na różnych danych.
- Jeśli wątki rozgałęziają się (np. if), GPU wykonuje je sekwencyjnie — to tzw. divergence i powinniśmy tego unikać.

Współbieżność na GPU

Iloma rdzeniami dysponują przykładowe modele

Model	Rdzenie CUDA	Rdzenie Tensor (5 gen)	Rdzenie RT (4 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

Ile wątków potrafią obsłużyć? Czym jest WARP

- Sprzętowo jest to układ scalony karty graficznej posiadający fizyczne jednostki zarządzające zwane *Warp Schedulers*. W jednym cyklu zegara pobierają i wysyłają do wykonania instrukcję tylko dla paczki składającej się dokładnie z 32 wątków
- Działa według zasady SIMT

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

Ile wątków jednocześnie?

- Liczba aktywnych WARPów zależy od liczby SM (Streaming Multiprocessor)
- Przykładowo w RTX serii 50 każdy SM obsługuje 48 WARPów

Model	Liczba bloków SM	Maks liczba warp jednocześnie	Łączna liczba przetwarzanych wątków
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

Współbieżność na GPU

Mamy 3 drogi

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

CUDA

CUDA to platforma firmy NVIDIA do obliczeń przyspieszanych sprzętowo, zapewniająca warstwę oprogramowania, która umożliwia aplikacjom wykorzystanie mocy obliczeniowej procesorów graficznych (GPU). Programiści mogą tworzyć kod w językach takich jak C++, Python i Fortran lub korzystać z bibliotek i frameworków przyspieszanych przez GPU, na przykład PyTorch. Taka elastyczność pozwala na integrację obliczeń GPU z dowolną warstwą stosu technologicznego, co przekłada się na optymalną funkcjonalność i wydajność.

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

CUDA Toolkit + wrapper

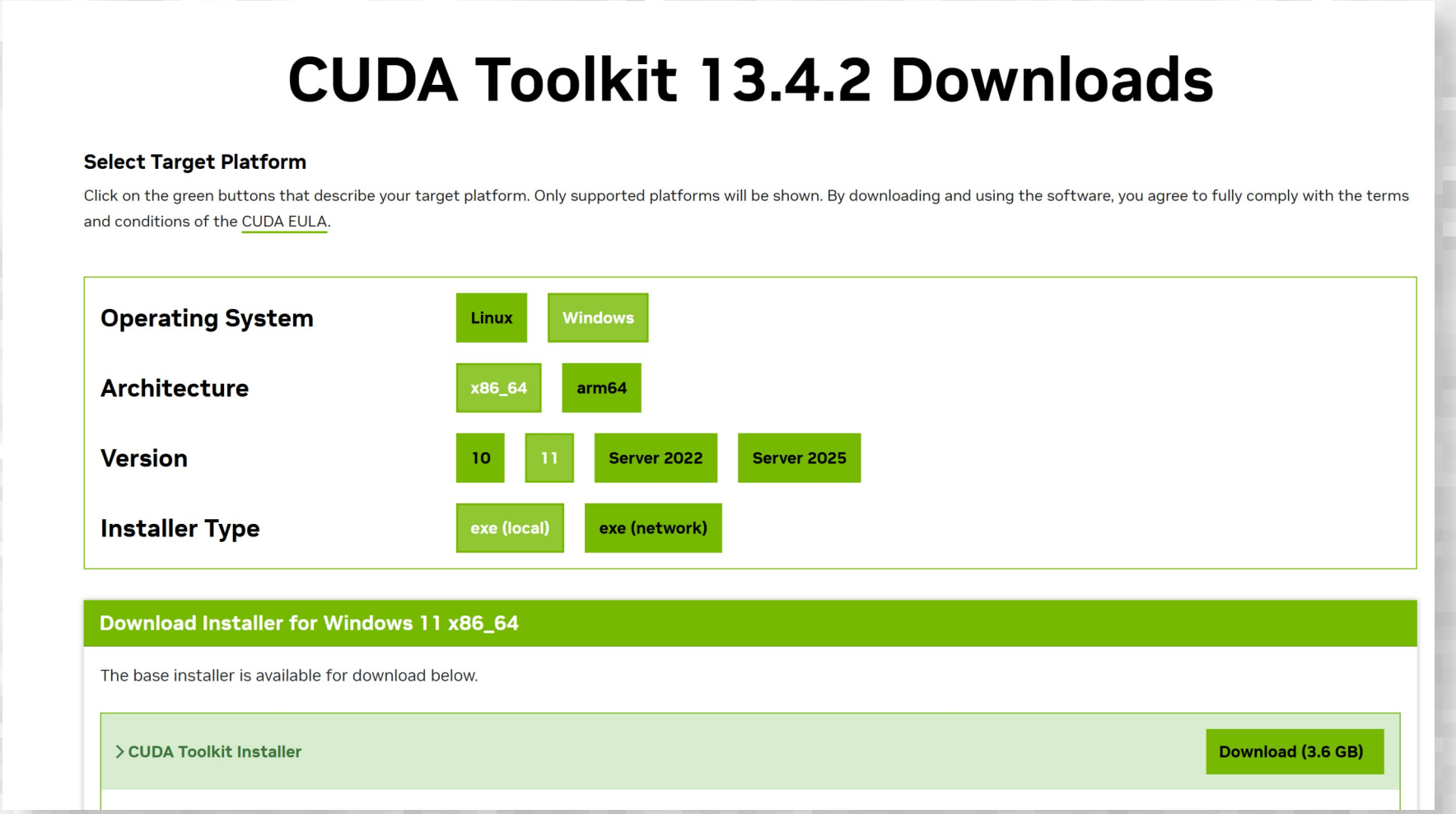
CUDA Toolkit – kluczowy element platformy CUDA – dostarcza kompilator, biblioteki oraz narzędzia programistyczne niezbędne do tworzenia aplikacji wykorzystujących procesory graficzne.

ILGPU - kompilator JIT GPU dla .NET: https://ilgpu.net/docs/

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

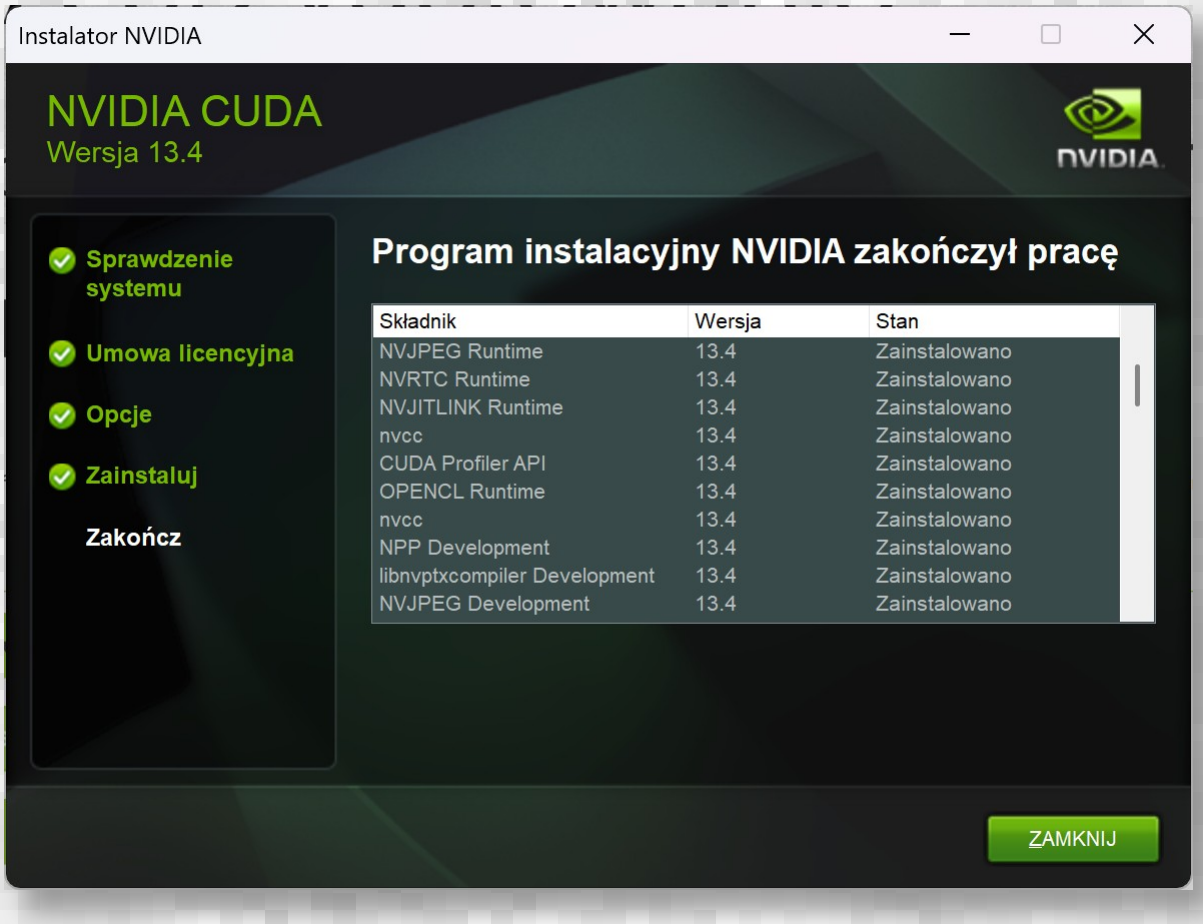
CUDA przygotowanie

Zaczynamy od pobrania i zainstalowania CUDA ToolKIT



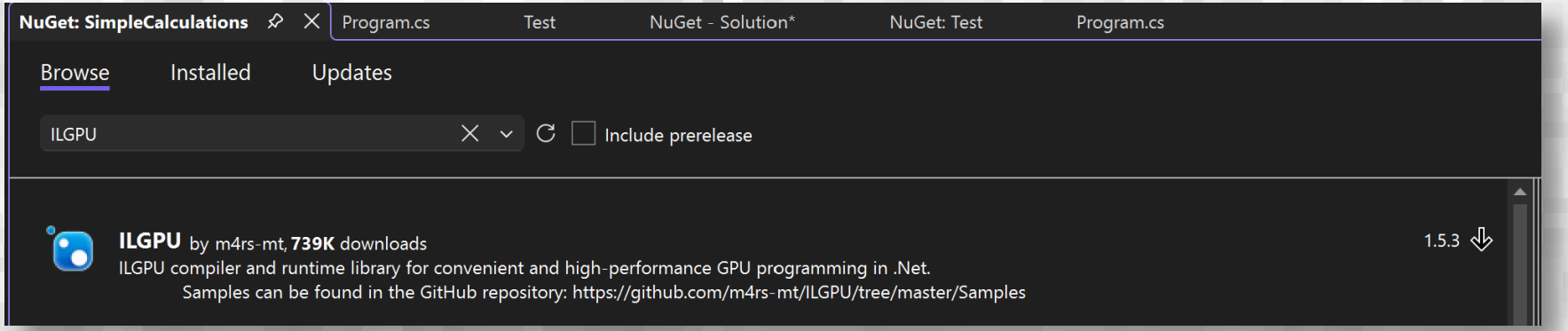
CUDA przygotowanie

Finalnie powinniśmy mieć listę składników



CUDA Pierwszy program

- W nowym projekcie dodajemy nugetem ILGPU



Dodajemy prosty kod metody kernela (czyli tego co jest uruchamiane na GPU

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


ConcurrentGPU/SimpleCalculations

CUDA Proste operacje

- Tworzymy obiekty kontekstu i akceleratora

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

- Inicjalizujemy zmienne i dane wejściowe

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

- Proste operacje na CPU (dla porównania)

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


ConcurrentGPU/SimpleCalculations

CUDA proste operacje

- Przygotowujemy bufor do GPU

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

- Przekazujemy dane do GPU oraz przygotowujemy akcję kernela

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

- Wykonanie obliczeń i synchronizacja

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

- Kopiowanie z powrotem do bufora

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

CUDA wynik

Wynik dla 10 000 000 elementów:

```
CPU time: 10 ms

GPU kernel only: 2 ms

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

Nie tego się spodziewaliśmy

CUDA Cięższe obliczenia

- Metoda bardziej obciążająca

```
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;
}
```

- Dodatkowo wersja na CPU w wersji współbieżnej

```
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 wynik

Wynik dla 10 000 000 elementów:

```
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
```

Teraz widać różnicę

Kernel

- Kernel jest podstawowym programem wykonywanym na karcie graficznej. Patern jest bardzo podobny do *Parallel.For* tzn:
- Mamy indeks startowy i końcowy
- Mamy dostęp do jakiejś kolekcji elementów
- Wykonujemy kawałek kodu dla konkretnego elementu o podanym indeksie w kolekcji.
- Kolejność wykonywania nie jest deterministyczna.

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

Pamięć

- Gdy komputery były jeszcze 8 bitowe to procesor potrafił być wolniejszy od Pamięci.
- Teraz pamięć jest tym wąskim gardłem obliczeń. Dzieje się to przede wszystkim z ograniczeń fizycznych. Im dalej komórki pamięci są oddalone od procesora tym więcej czasu potrzebuje sygnał na dotarcie z/do pamięci.
- Dlatego szybka pamięć to ta co jest wbudowana w procesor, np. Rejestry.
- Dlatego w procesorach stosuje się różne sztuczki np. *prefetching* (pobieranie danych do cache procesora)

Pamięć

Przepustowość:

- DDR5-6400 to 51.2 GB/s na kanał, czyli w trybie dual-channel 102.4 GB/s
- RTX 5080 GDDR7 to 960GB/s
- Nvidia H100 z HBM3 to już około 3000GB/s

Przewaga na korzyść kart graficznych jest nawet kilkudziesięcio krotna.

Context

- Wszystkie klasy i funkcje biblioteki ILGPU korzystają z instancji *ILGPU.Context*.
- Zadaniem kontekstu jest przede wszystkim pełnienie roli interfejsu dla kompilatora ILGPU.
- Przechowuje on również pewien stan globalny.
- Domyślnie tworzymy przez

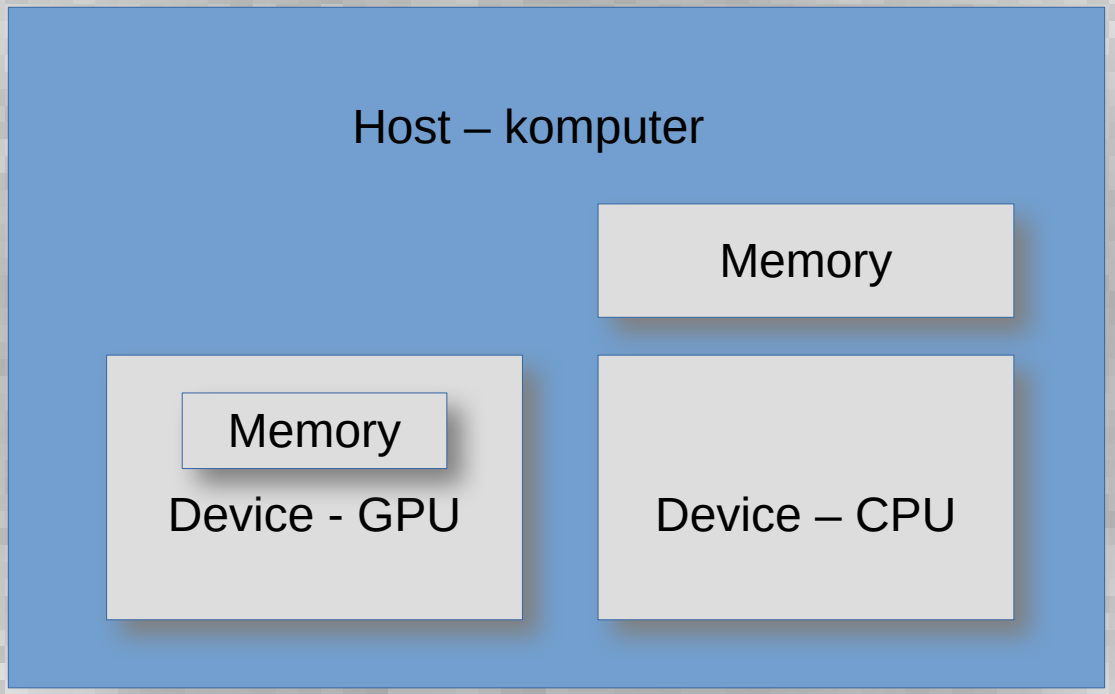
```
using Context context = Context.CreateDefault();
```

- Context jest *IDisposable* więc powinniśmy wywoływać Dispose lub umieszczać w bloku using

Akceleratory

- W ILGPU akcelerator reprezentuje sprzętowy lub programowy procesor graficzny (GPU).
- Każdy program w ILGPU wymaga co najmniej jednego akceleratora.
- Obecnie dostępne są następujące typy akceleratorów:
  - CPU, `using CPUAccelerator accelerator = context.CreateCPUAccelerator(0);`
  - CUDA, `using CudaAccelerator accelerator = context.CreateCudaAccelerator(0);`
  - OpenCL, `using CLAccelerator accelerator = context.CreateCLAccelerator(0);`

Bufory



W *ILGPU* o pamięci mówi się w kategoriach tablic, nawet gdy chcemy przekazać pojedynczą wartość to przekazujemy tablice jednoelementową.

MemoryBuffer1D<T>

- **MemoryBuffer** to kopia pamięci przydzielonej na urządzeniu (device), dostępna po stronie hosta. W istocie jest to po prostu wskaźnik do obszaru pamięci zaalokowanego na urządzeniu.
- Obiekt ten zawsze uzyskuje się za pośrednictwem akceleratora (*Accelerator*).
- Wymaga użycia przestrzeni nazw:

```
using ILGPU.Runtime;
```

- Przykładowy sposób tworzenia:

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

Copy From/To CPU

- Po przydzieleniu obiektu MemoryBuffer ładujemy dane:

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

- Lub je pobieramy:

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

ArrayView<T>

- *ArrayView* to kopia pamięci, znajdująca się po stronie urządzenia (GPU), która została przydzielona z poziomu hosta (CPU).
- Jest to element API *MemoryBuffer*, z którym bezpośrednio współpracują kernele oraz układ GPU.
- Obiekt ten zawsze uzyskuje się za pośrednictwem *MemoryBuffer*.
- Wymaga: `using ILGPU.Runtime;`
- Podstawowy sposób tworzenia: `ArrayView1D<int, Stride1D.Dense> ints = bufferOnDevice.View;`
- Wewnątrz kernela obiekt *ArrayView* zachowuje się dokładnie tak, jak można by oczekiwać od zwykłej tablicy.

Przykłady z memory

```
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 – główna metoda

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

- Można alokować i przekazywać tylko nienulowalne typy wartościowe o ustalonej wielkości
- Brak klas
- Brak referencji
- Brak struktur o dynamicznym rozmiarze
- Pierwszym parametrem kernela musi być jego indeks.
- Kernel zawsze iteruje w określonym zakresie, może to być 1, 2 lub 3 wymiary
- Funkcja może zawierać dowolną logikę ale unikajmy warunkowych rozgałęzień

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];
        }
```

Wyniki

10 prób, wyniki w ms. Jak widać GPU nie pokazało pełnej mocy

Próba	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
średnia	**215,2**	**179,8**	**85,6**	**389,7**

CUDA vs SIMD vs Scalar

runda druga

Wykonajmy teraz podobny test dla nieco bardziej wymagających obliczeń

```
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 nowy zawodnik

- Kompilator teoretycznie sam dobiera najbardziej optymalne rozwiązanie do zadanych obliczeń, ale możemy też sami zdecydować, które instrukcje procesora wykorzystamy.
- Do tego wykorzystamy klasę *AVX2* z jej zoptymalizowanymi metodami
- Problem polega na tym, że metody *Avx.LoadVector256* i *Avx.Store* przyjmują tylko wskaźnik (*float**) do danych w pamięci. Dlatego trzeba użyć ***unsafe***

AVX2 nowy zawodnik

```
 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 nowy zawodnik

- Alternatywnie zamiast dwóch metod Add i Multiply,

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

- możemy użyć dedykowanej jednej *MultiplyAdd*

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

Porównanie

Dziesięć prób, wyniki w ms

Próba	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
średnia	**32296,1**	**9217,1**	**9233,6**	**6296,5**	**363,1**	**174,8**

Jeżeli różnica między GPU i FMA P dla 1024 iteracji wydaje się niewielka, to poniżej przykład dla 256*1024

FMA P: 168039 ms

GPU kernel only: 148 ms

GPU total (incl. transfers): 378 ms

Krótkie zestawienie SIMD

Instrukcje	Co robią
FMA — Fused Multiply-Add (Fma.MultiplyAdd)	Wykonuje v = v*m + a
Avx.Multiply / Avx.Add / Avx.Subtract / Avx.Divide	Wykonuje podstawowe operacjie SMID na 8 floatach na raz
Avx.Sqrt	Wektorowy pierwiastek kwadratowy
Vector<float>.SquareRoot	j.w. ale przy dłuższych pętlach mniej wydajny
Avx.And / Avx.Or / Avx.Xor / Avx.AndNot	Bitowe operacje SIMD
Avx.Compare / Avx.CompareGreaterThan / Avx.CompareLessThan	Porównują 8 floatów naraz i zwracają maskę bitową
Avx.Blend / Avx.BlendVariable	Łączą dwa wektory na podstawie maski
Avx.Permute / Avx.Shuffle	Przestawiają elementy wektora
Avx.HorizontalAdd / HorizontalSubtract	Dodają lub odejmują elementy wektora w poprzek.
Avx.BroadcastScalarToVector256	Powiela jedną wartość na cały wektor.

Krótkie zestawienie SIMD

Instrukcje	Co robią
Avx.LoadVector256 / Avx.Store	Ładują i zapisują 8 floatów naraz.
Avx.MaskLoad / Avx.MaskStore	Ładują/zapisują tylko wybrane elementy wektora.
Avx2.GatherVector256	Ładuje elementy z nieciągłych adresów (indeksowanie wektorowe).
Avx2.PermuteVar8x32	Zaawansowane permutacje elementów
Avx2.ShiftLeft / ShiftRight	Przesunięcia bitowe SIMD.
Avx2.MultiplyLow / MultiplyHigh	Mnożenie integerów z zachowaniem dolnych/górnych bitów.
Avx2.AddSaturate / SubtractSaturate	Dodawanie/odejmowanie z saturacją (bez przepełnienia).
Avx2.CompareEqual / CompareGreaterThan	Porównania integerów SIMD.
Avx2.Blend / BlendVariable	Łączenie wektorów na podstawie maski.
Avx2.Extract / Insert	Wyciągają lub wstawiają elementy do wektora.
Avx2.Pack / Unpack	Pakują lub rozpakowują dane (np. 16-bit → 8-bit).

Mnożenie macierzy

- Jeden wątek

```
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

Mnożenie macierzy

- Wersja CPU współbieżna

```
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();
```

Mnożenie macierzy

- 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();
```

Mnożenie macierzy

- 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;
}
```

Mnożenie macierzy

- GPU Tiled (omówione dalej)

```
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);
```

Mnożenie macierzy

- 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
```

Mnożenie macierzy

- GPU Tiled

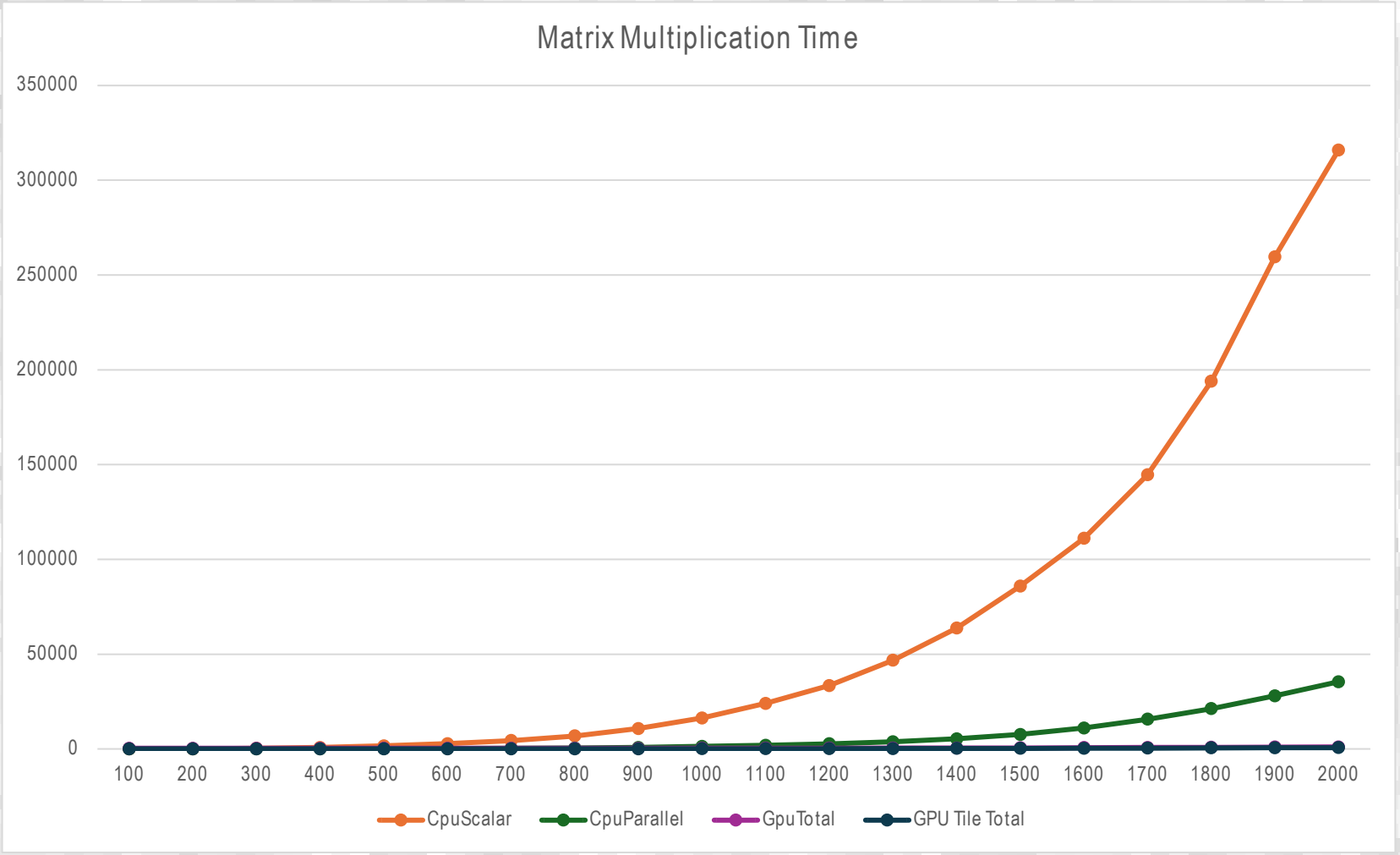
```
--->…
   for (int t = 0; t < numTiles; t++)
     {
         int aCol = t * TILE + tx;
         int bRow = t * TILE + ty;
         // guarded loads
         sharedA[ty * TILE + tx] = (row < N && aCol < N) ? A[row, aCol] : 0.0f;
         sharedB[ty * TILE + tx] = (bRow < N && col < N) ? B[bRow, col]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

Mnożenie macierzy

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

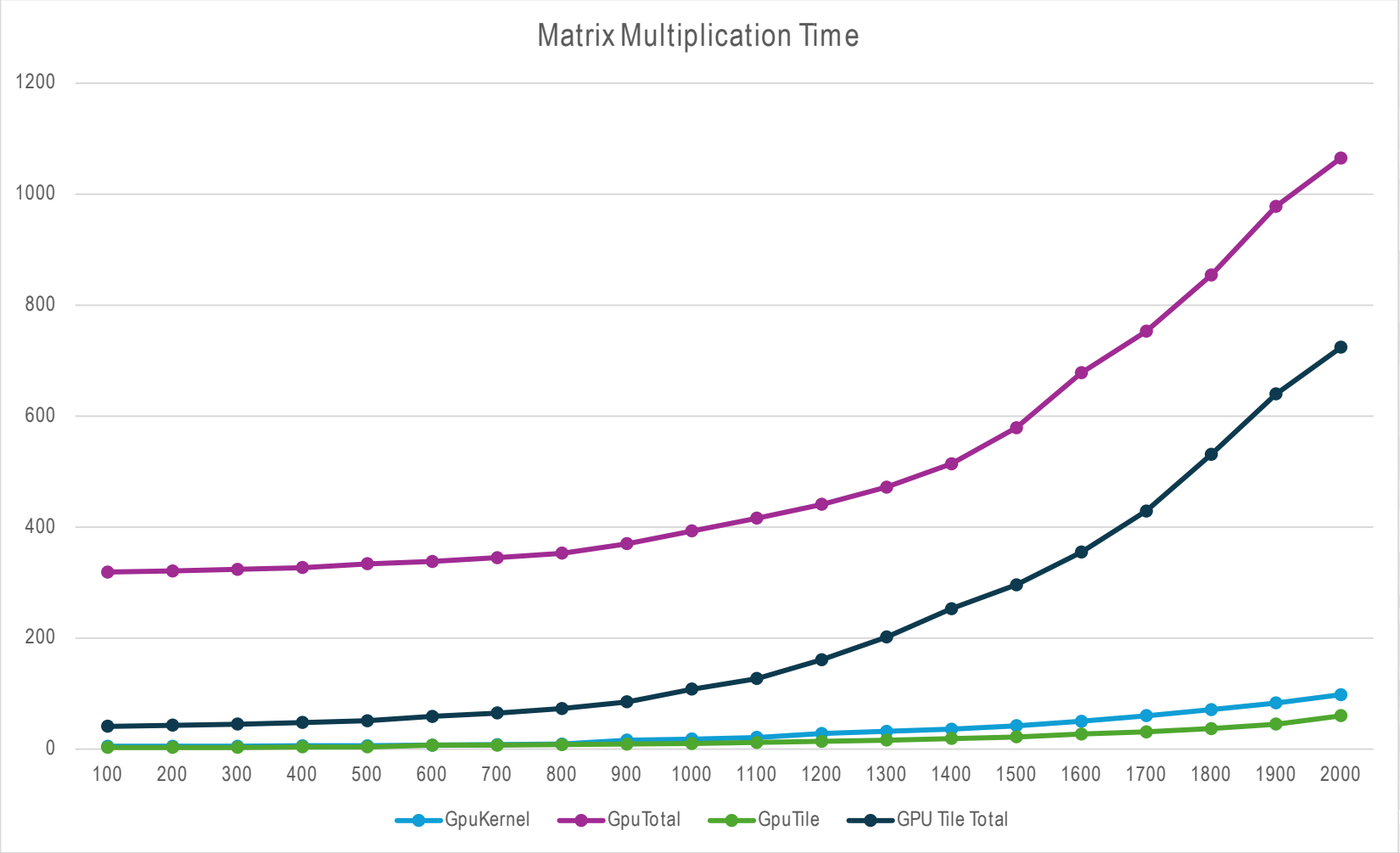
Mnożenie macierzy

- Wyniki CPU single, CPUP, GPU i GPU Tile



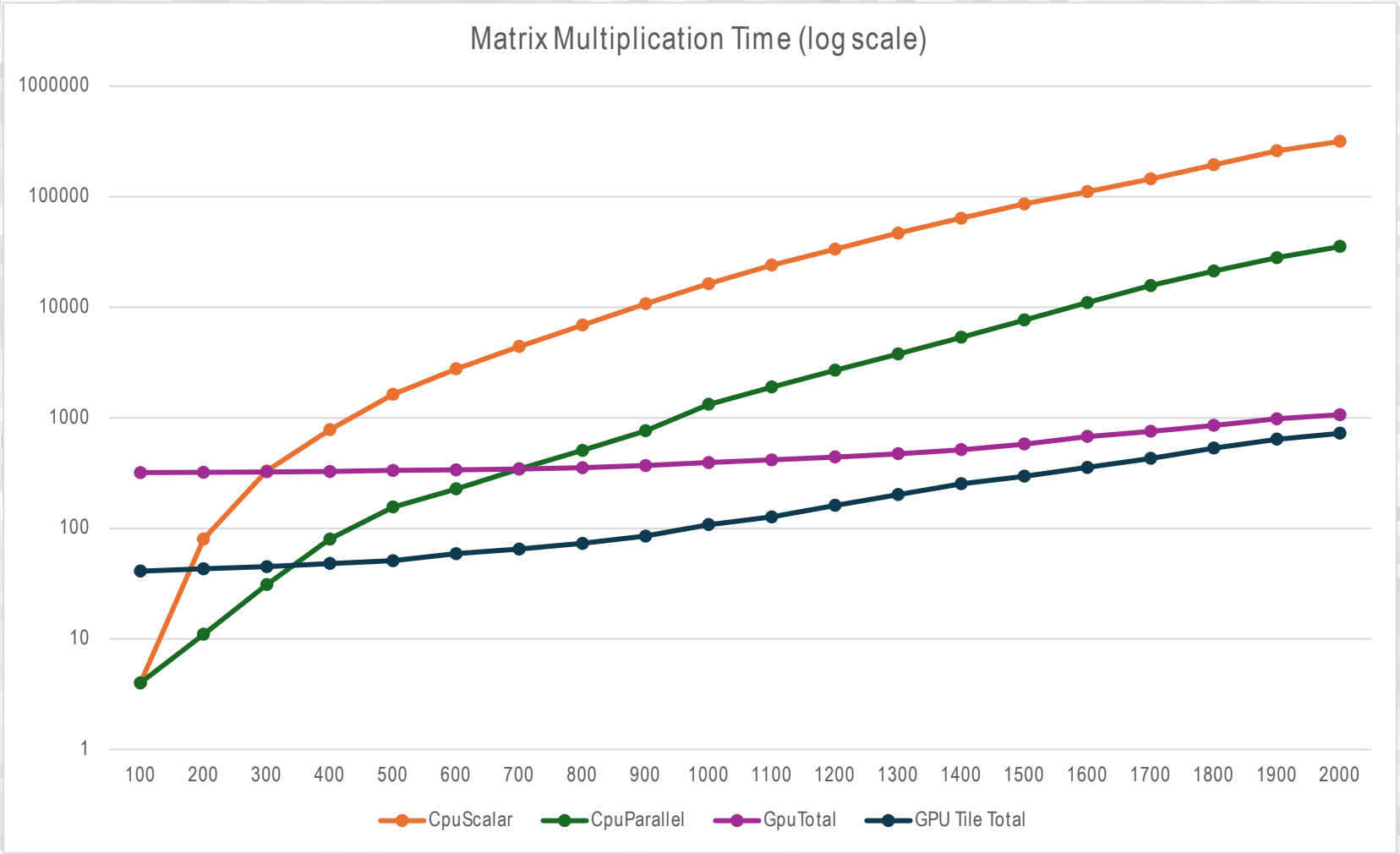
Mnożenie macierzy

- Tylko GPU vs GPU Tile



Mnożenie macierzy

- Wyniki CPU single, CPUP, GPU i GPU Tile



Struktury

- Na GPU nie zawsze możemy używać danych identycznie jak na CPU używając C#. Dlatego ILGPU nakłada pewne ograniczenia związane z kodowaniem niższego poziomu.
- Przykładowo nie możemy używać struktur wewnątrz klas.

ConcurrentGPU/NbodyExample NBodyWPFExample

Struct of Arrays vs Array of Structs

- Załóżmy że mamy strukturę Car z podstawowymi parametrami

```
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;
}
```

- Po utworzeniu i zainicjowaniu tablicy tych struktur przekazujemy je do GPU

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


ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- Następnie uruchamiamy obliczenia na GPU

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

- Gdzie mamy wykonywane jakieś proste wyliczanie pozycji

```
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;
}
```

- Po wykonaniu obliczeń zwracamy dane do hosta za pomocą *CopyToCPU*

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- Po 10 próbach z 1mln samochodów i 100 iteracjach wyniki mamy następujące (w ms):

Próba	InitializationTime	PreparationTime	KernelTime	ReturningTime	Suma
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
Średnia	**2645,9**	**1263,8**	**64,5**	**676,4**	**4650,6**

Struct of Arrays vs Array of Structs

- Dla porównania 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];
```

- Alokujemy na GPU każdą tablicę z osobna

```
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

- Przekazujemy do GPU tylko to co nam potrzebne

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

- Metoda kernela robi te same obliczenia tylko korzysta z 3 osobnych tablic

```
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

- Po 10 próbach z 1mln samochodów i 100 iteracjach wyniki mamy następujące (w ms):

Próba	InitializationTime	PreparationTime	KernelTime	ReturningTime	Suma
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
Suma	**2162**	**972,6**	**33**	**214**	**3381,6**

Struct of Arrays vs Array of Structs

- Spróbujmy tego samego z odchudzoną wersją Car

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

}
```

- Cześć danych nie jest potrzebna do obliczeń

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- Po 10 próbach z 1mln samochodów i 100 iteracjach Wersja AoS odchudzona. (w ms):

Próba	InitializationTime	PreparationTime	KernelTime	ReturningTime	Suma
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
Średnia	**1261,5**	**312,3**	**22,9**	**236,4**	**1833,1**

ConcurrentGPU/CarsAoS

Struct of Arrays vs Array of Structs

- W wersji SoA dla porównania zrezygnujemy z dodatkowych tablic:

```
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];
```

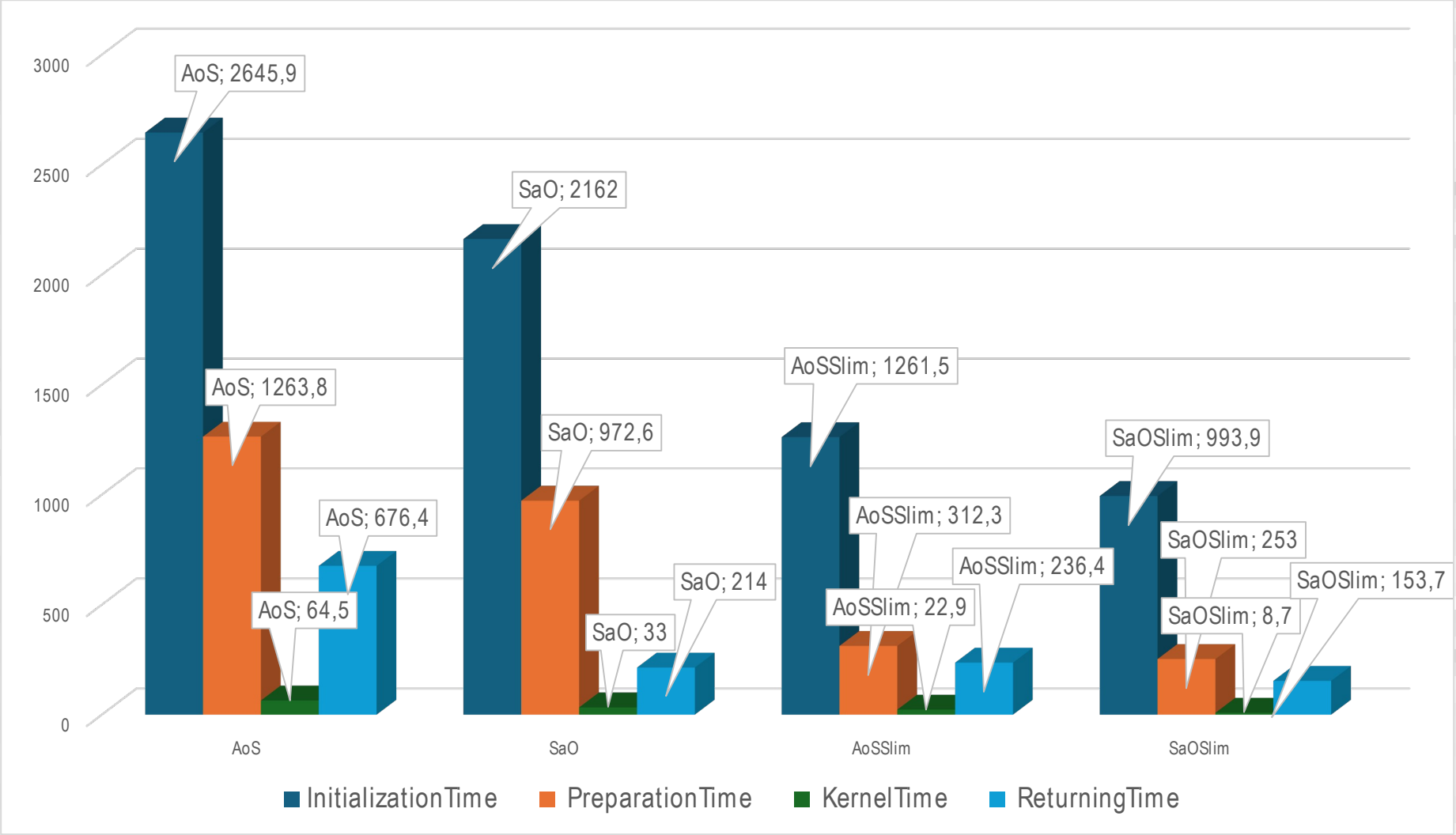
- *velocity* nie musi być przekazywane powrotnie jeśli nie zmieniamy.

Struct of Arrays vs Array of Structs

- 10 prób, 1mln samochodów, 100 iteracji (wersja odchudzona) wyniki w ms.

Próba	InitializationTime	PreparationTime	KernelTime	ReturningTime	Suma
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
Średnia	**993,9**	**253**	**8,7**	**153,7**	**1409,3**

Struct of Arrays vs Array of Structs



Nbody – Kolejny przykład

```
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)
     { ... }
 }
```

Struktura reprezentująca cząstkę wraz z metodami które służą do aktualizacji pozycji, przyspieszenia i prędkości

ConcurrentGPU/NbodyExample NBodyWPFExample

Nbody – Kolejny przykład

Osobne struktury

```
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;
    }
```

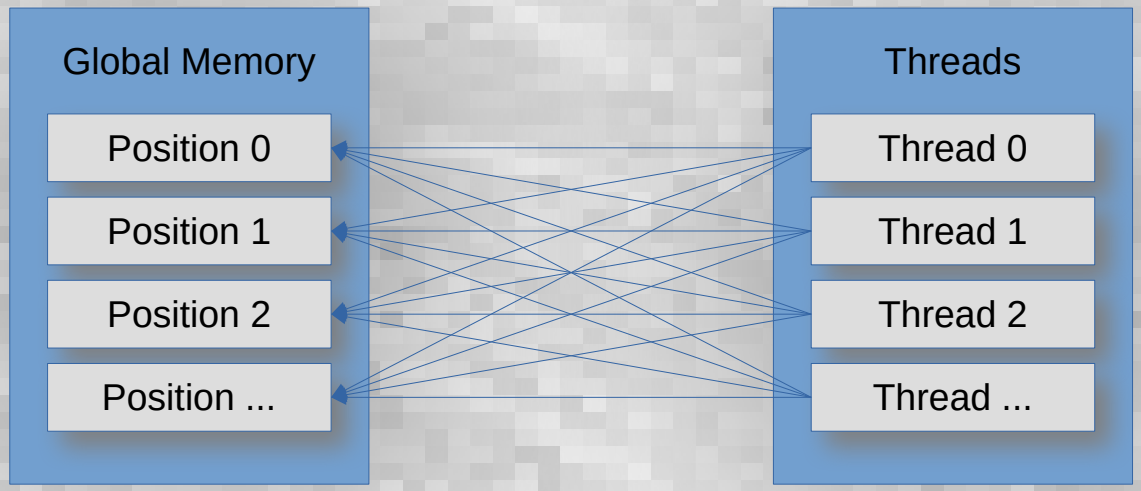

ConcurrentGPU/NbodyExample NBodyWPFExample2

Nbody – Tiled

- W algorytmie N-Body każda cząstka musi odczytać pozycję wszystkich pozostałych cząstek. Dla 32 768 cząstek:

$$32\,768 \times 32\,768 \approx 1 \text{ miliard interakcji}$$

- Bez tilingu każdy wątek wielokrotnie odczytuje dane z globalnej pamięci GPU:

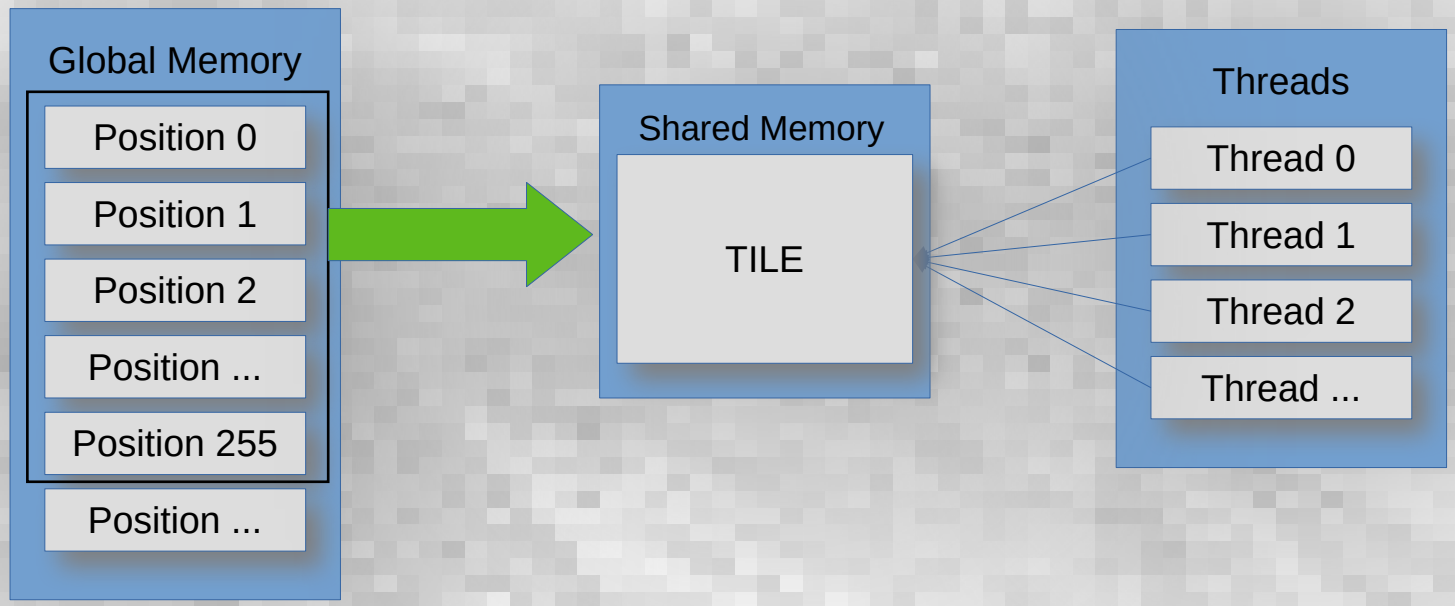


- Pamięć globalna ma dużą przepustowość, ale również wysokie opóźnienia.
- Te same dane są wielokrotnie odczytywane przez różne wątki.

ConcurrentGPU/NbodyExample NBodyWPFExample3

Nbody – Tiled

Grupa wątków ładuje fragment danych do szybkiej pamięci współdzielonej (Shared Memory).



Wczytaj, zsynchrnizuj (Group.Barrier()), wykonaj obliczenia wykorzystując shared memory, załaduj kolejny fragment.

ConcurrentGPU/NbodyExample NBodyWPFExample3

Nbody – Tiled

Dodatkowa modyfikacja to rozbicie obliczeń na dwa osobne kernele:

`ParticleKernelTiled`
`RenderKernel`

Kluczowe elementy:

```
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 – Podsumowanie

V1 średnio około 20 fps

V2 około 52 fps

V3 około 100 fps

Koniec

Dziękuję za uwagę