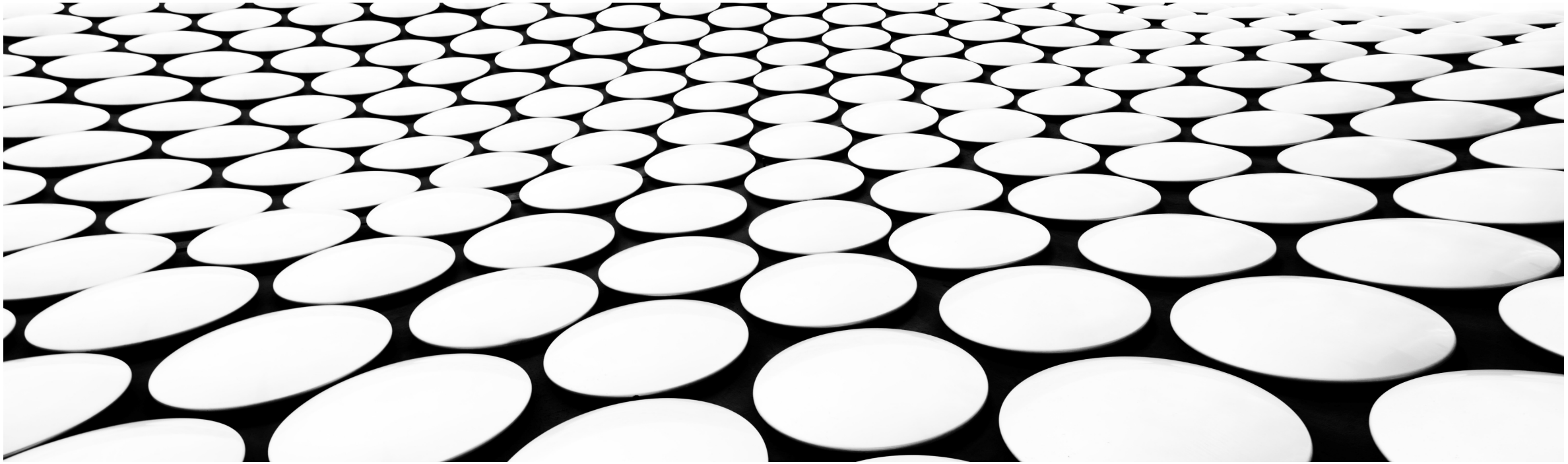
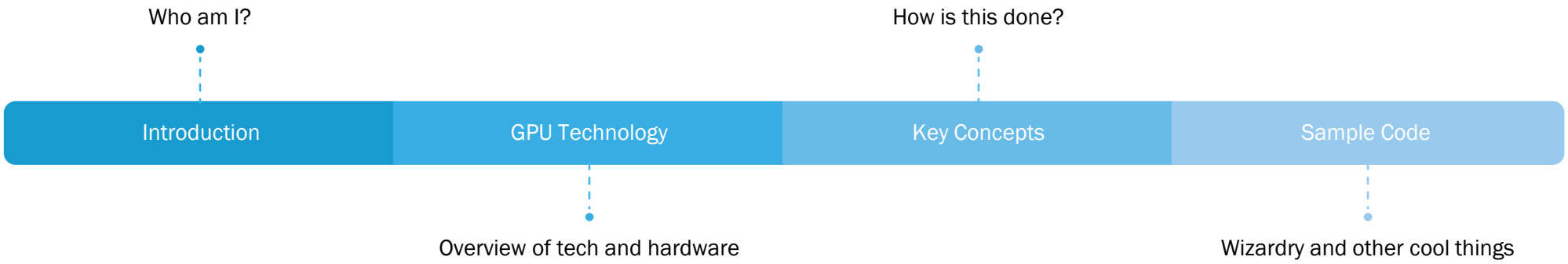
GPU PROCESSING WITH C#

INTRODUCTION TO ILGPU (PART ONE)

Christopher Aliotta
chrisaliotta@quantalytix.com
https://quantalytix.com



DOTNET MEETUP AGENDA

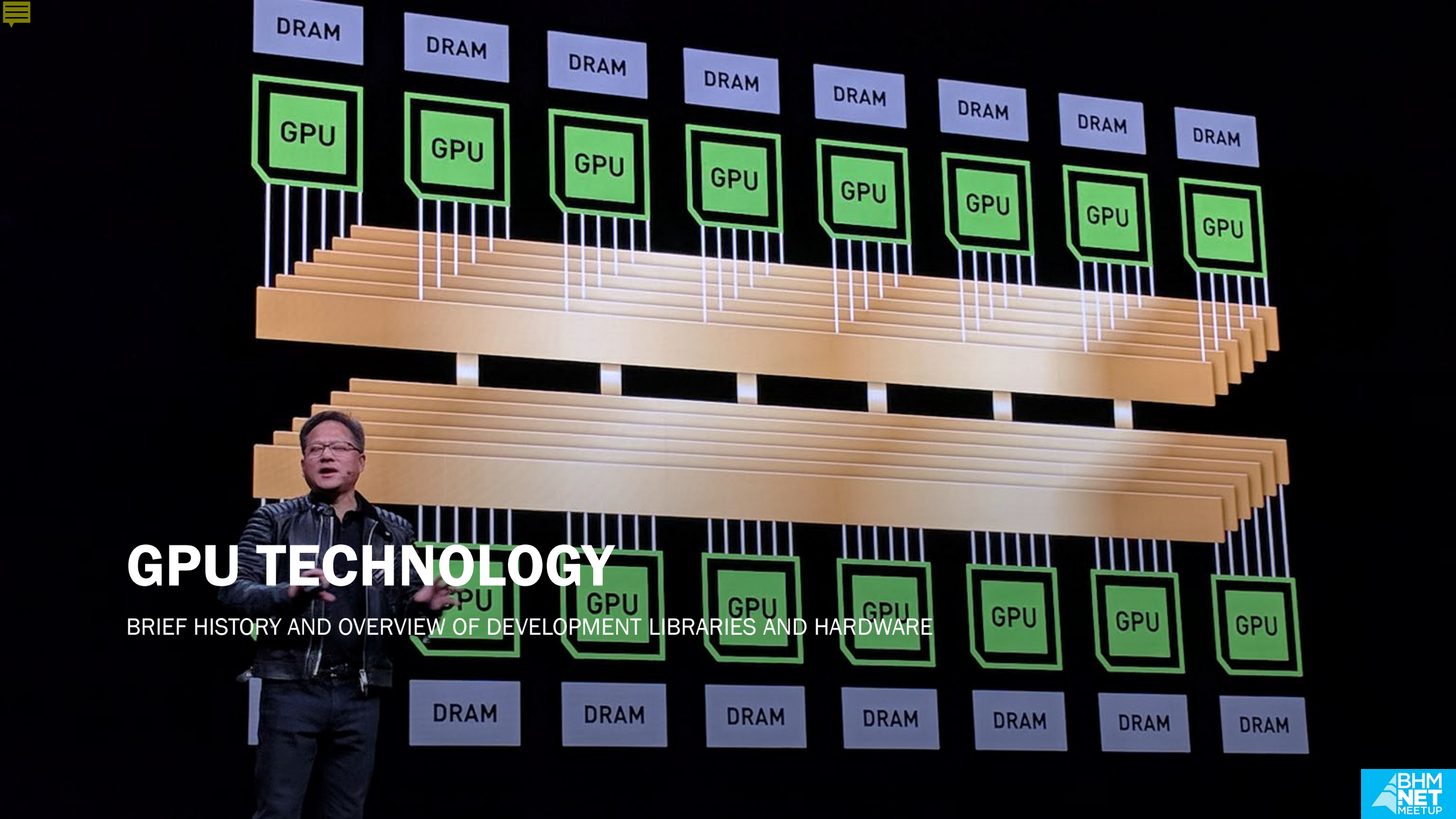


WHO AM I?

- Software developer for over 20+ years with experience programming.
- Former banker for 10 years with a focus on quantitative modeling and simulations.
- Co-founder of Quantalytix, a FinTech startup based out of Innovation Depot.
- Father and husband; enjoys playing D&D, Transit, and EVE Online.
- Hobbyist hardware developer.

I AM STILL LEARNING...

(So go easy on me)



GPU TECHNOLOGY

BRIEF HISTORY AND OVERVIEW OF DEVELOPMENT LIBRARIES AND HARDWARE

BHM .NET MEETUP

GPU TECHNOLOGY

- The CPU has always been slow for graphics processing.
- Graphics processing solved for slow CPU processing by streamlining inherently parallel tasks.
- In 2006, GPUs became programmable with Nvidia's release of CUDA.

WHAT PROBLEMS DOES IT SOLVE?

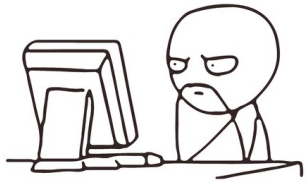
- Computing large amounts of data in parallel.
- It is used in complex graphics pipelines as well as scientific computing; more so in fields with large data sets like genome mapping.
- Cryptographic currencies (and password dehashing).
- Machine learning.

CUDA AND OPENCL

- CUDA is an acronym for Compute Unified Device Architecture. Proprietary to Nvidia and will only work on Nvidia devices.
- OpenCL (Open Computing Language) is the main competitor to CUDA as it is an open, royalty-free, library supported by the Khronos group (OpenGL, etc.).
- We'll be focusing mostly on CUDA implementations for this discussion.

LIBRARIES, MIDDLEWARE, LANGUAGES

- Abundance of languages that support CUDA.
- For some reason C# does not get much love…



Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	 Embedded	 Consumer Desktop/Laptop	 Professional Workstation	 Data Center

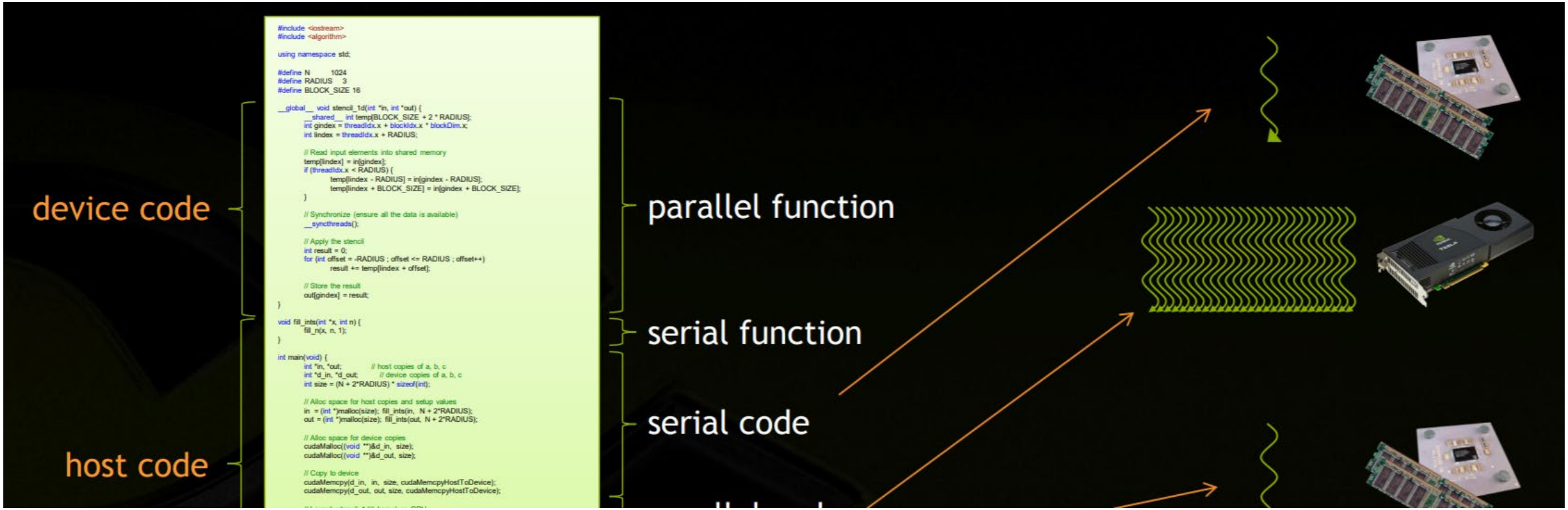
TODAY'S HARDWARE

- Most of the hardware needed can be found on any one of the leading cloud platforms.
- The Nvidia A100 starts at $12,500.
- Good news, you can still leverage GPU processing on your old Nvidia GTX!

HARDWARE PERFORMANCE

- Floating point operations measured in Teraflops.
- Modern CPUs are somewhere in the range of 5-7 Gigaflops.
- 1 TFLOPS is equivalent to 1000 GFLOPS.
- GPUs perform in orders of magnitude larger than CPUs in floating point operations.

	NVIDIA A100 for HGX	NVIDIA A100 for PCIe
Peak FP64	9.7 TF	9.7 TF
Peak FP64 Tensor Core	19.5 TF	19.5 TF
Peak FP32	19.5 TF	19.5 TF
Peak TF32 Tensor Core	156 TF | 312 TF*	156 TF | 312 TF*
Peak BFLOAT16 Tensor Core	312 TF | 624 TF*	312 TF | 624 TF*
Peak FP16 Tensor Core	312 TF | 624 TF*	312 TF | 624 TF*
Peak INT8 Tensor Core	624 TOPS | 1,248 TOPS*	624 TOPS | 1,248 TOPS*
Peak INT4 Tensor Core	1,248 TOPS | 2,496 TOPS*	1,248 TOPS | 2,496 TOPS*
GPU Memory	40 GB	40 GB
GPU Memory Bandwidth	1,555 GB/s	1,555 GB/s
Interconnect	NVIDIA NVLink 600 GB/s** PCIe Gen4 64 GB/s	NVIDIA NVLink 600 GB/s** PCIe Gen4 64 GB/s
Multi-instance GPUs	Various instance sizes with up to 7MIGs @5GB	Various instance sizes with up to 7MIGs @5GB

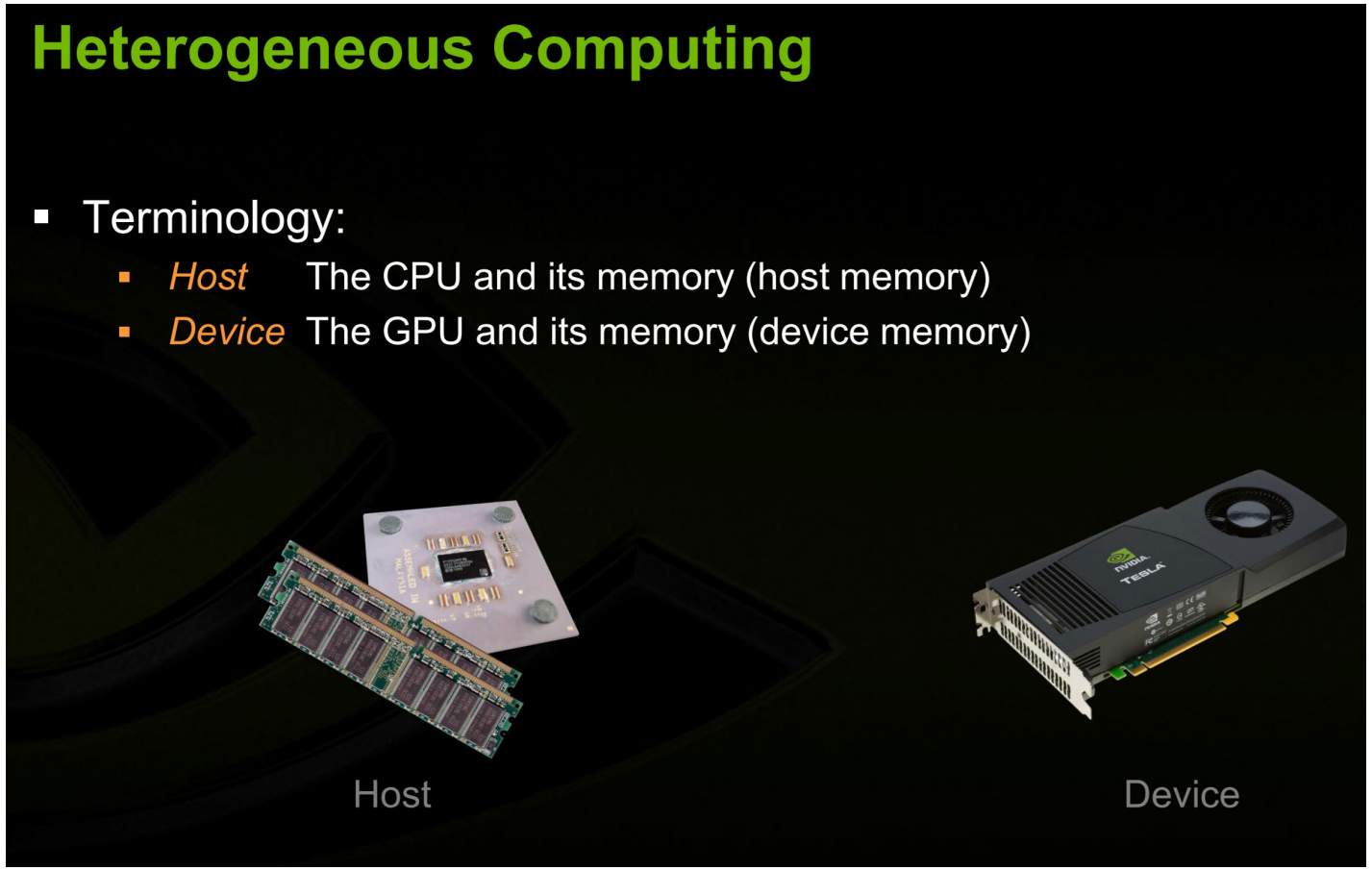


KEY CONCEPTS

Development related concepts for getting started!

ANATOMY OF A CUDA PROGRAM

- Device code vs. Host code
- Device code is segmented into what are called **Kernels**.
- CUDA originally based on C, C99 standard.
- Incorporated C++ standards in 2008 due to high developer demand.



KERNELS

Standard C Code

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
```

Parallel C Code

```
__global__
void saxpy_parallel(int n,
                    float a,
                    float *x,
                    float *y)
{
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

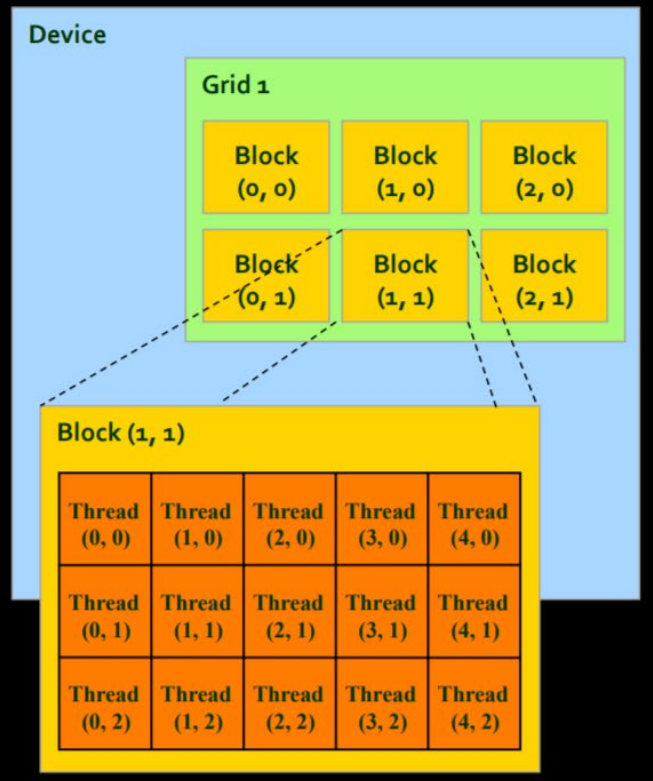
GRIDS, BLOCKS, THREADS

GPU MULTIDIMENSIONALITY

- A **Grid** is a composition of **Blocks**.
- **Blocks** are a composition of **Threads**.
- All of these can be represented in scalar or in 2D or 3D vectors.
- Block and thread count determined by the device.

Thread and Block ID and Dimensions

- Threads
  - 3D IDs, unique within a block
- Thread Blocks
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables
  - `threadIdx`, `blockIdx`
  - `blockDim`, `gridDim`
- Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads



SAMPLE CODE (C/C++)

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                     // host copies of a, b, c
    int *d_a, *d_b, *d_c;         // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);

     // Copy inputs to device
     cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
     cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

     // Launch add() kernel on GPU
     add<<<N/THREADS_PER_BLOCK THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

     // Copy result back to host
     cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

     // Cleanup
     free(a); free(b); free(c);
     cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
     return 0;
 }
```

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

OK... THIS IS GREAT, BUT WHAT ABOUT C#?

C# BASED CUDA LIBRARIES

- **managedCuda** (Open Source): managedCuda is the right library if you want to accelerate your .net application with Cuda without any restrictions. As every kernel is written in plain CUDA-C, all Cuda specific features are maintained. Even future improvements to Cuda by NVIDIA can be integrated without any changes to your application host code.
- **Alea GPU** (Proprietary): With Alea GPU, you can take advantage of this processing power to accelerate .NET and Mono applications in a simple and efficient way on Windows, Linux and Mac OS X. You develop your GPU code with the .NET language and tools you already know. The Alea GPU runtime system efficiently handles execution on the GPU and all the memory management.
- **ILGPU** (Open Source):ILGPU is a new JIT (just-in-time) compiler for high-performance GPU programs (also known as kernels) written in .Net-based languages. ILGPU is completely written in C# without any native dependencies which allows you to write GPU programs that are truly portable. It combines the convenience of C++ AMP with the high performance of CUDA. Functions in the scope of kernels do not have to be annotated (e.g. default C# functions) and are allowed to work on value types. All kernels (including all hardware features like shared memory, atomics and warp shuffles) can be executed and debugged on the CPU using the integrated multi-threaded CPU accelerator. And the best feature: it's free! ILGPU is released under the University of Illinois/NCSA Open Source License.

ILGPU

The BEST Implementation
IMNSHO

ILGPU: HOW IT WORKS


```
.visible .func (.param .s32 __cudaretf__Z15getCurThreadIdxv) _Z15getCurThreadIdxv ()
{
.reg .u32 %r<7>;
.loc    16  7   0
$LDWbegin__Z15getCurThreadIdxv:
.loc    16  8   0
mov.u32     %r1, %tid.x;
mov.u32     %r2, %ctaid.x;
mov.u32     %r3, %ntid.x;
mul.lo.u32  %r4, %r2, %r3;
add.u32     %r5, %r1, %r4;
st.param.s32    [__cudaretf__Z15getCurThreadIdxv], %r5;
ret;
$LDWend__Z15getCurThreadIdxv:
} // _Z15getCurThreadIdxv
```

Current thread computation

Device Function

```
.entry _Z9fooKernelIiEvPKT_iPS0_ (
    .param .u32 __cudaparm__Z9fooKernelIiEvPKT_iPS0__inArr,
    .param .s32 __cudaparm__Z9fooKernelIiEvPKT_iPS0__num,
    .param .u32 __cudaparm__Z9fooKernelIiEvPKT_iPS0__outArr)
{
.reg .u32 %r<19>;
.reg .pred %p<4>;
.loc    16  12  0
$LDWbegin__Z9fooKernelIiEvPKT_iPS0_:
.loc    16  16  0
mov.u32     %r1, %ntid.x;
mov.u32     %r2, %ctaid.x;
mul.lo.u32  %r3, %r2, %r1;
mov.u32     %r4, %tid.x;
add.u32     %r5, %r4, %r3;
ld.param.s32    %r6, [__cudaparm__Z9fooKernelIiEvPKT_iPS0__num];
setp.le.s32     %p1, %r6, %r5;
@%p1 bra    $Lt_1_1282;
mul.lo.u32  %r7, %r5, 4;
mul.lo.u32  %r8, %r6, 4;
```

Device function has been inlined. Same computation as in device function.

Template Kernel

ILGPU converts .NET code to PTX, an assembly like language, that is accepted by CUDA devices.

CODE EXAMPLE