



EE 524 P

Applied High-Performance GPU Computing

LECTURE 1 : Wednesday, January 6, 2021

Instructor: Dr. Colin Reinhardt

EE Professional Masters Program

Winter 2021



Lecture 1 : Outline

- Welcome to (*new all-remote*) Applied GPU Computing
- About me
- Course schedule and logistics
- Why GPU computing today?
- Parallel execution on GPUs
- Exercise 1a
- Discuss HW-1



About Me



- 1st programmed Apple 2e (BASIC) and Commodore 64 – early 1980s
- Professional Software Engineer, Seattle (1997-2004)
- BSEE+BS Physics from UW (2005)
- PhD in EE from UW (2010)
 - Committee: Profs Kuga/Ritcey/Ishimaru/Riley
 - Dissertation work focused on FSO comms, radiative transfer theory, inverse problems
- ASEE/DOD SMART Doctoral Fellow (2007-2010)
- DOD/US Navy Research Engineer (2011-present)
 - EO/IR atmospheric propagation physics, modeling & simulation, characterization
 - Physics-based (3D+T) scene simulation, computer M&S, 3D CG, ray tracing
 - EO/IR systems and sensors, optics, imaging, image processing
 - thermal heat transfer
- Affiliate Assistant Professor, UW EE (2016-present)
- Affiliate/GOV Sponsor, APL-UW, AIRS Laboratory (2016-present)
- Fun: surfing, rock climbing, mountain biking, yoga, family



This Class: A bold new virtual frontier

➤ Previous classes

- held in EEB 365 computer lab
 - Intel SOC (Sky Lake, Kaby Lake) processor architectures
 - OpenCL

➤ This Winter-2021 class

- all online/remote
- CUDA and OpenCL, NVIDIA and Intel GPU architectures
- Cloud-based virtual workstations & GPU instances
 - AWS EC2 and AppStream

➤ Please bear with probable technical issues, glitches, etc.

- We're all doing our best!



Course “Philosophy”

- ▶ We'll study GPU computing via a series of “passes” (repeated revisits) with progressive “zoom” perspective and iterative refinements. Introduce more details/theory/tools and advanced GPU techniques with each pass. (and some background/history where relevant)
 - ▶ Hardware (HW) and system microarchitectural perspective
 - ▶ Software (SW): API, code and algorithms
 - ▶ Execution/Runtime analysis, profiling, debugging and supporting tools
 - ▶ theoretical
 - ▶ empirical
 - ▶ Advanced techniques: concurrency, synchronization, assembly language/ISAs, etc...
 - ▶ **Homeworks** will be used to build and reinforce the fundamentals
- ▶ Then, once we've build up sufficient breadth & depth, we'll switch to a couple of deep-dive studies of important & interesting current applications.
 - ▶ HPC/scientific
 - ▶ ML/DL
 - ▶ Along with working on a **final project** which pulls everything together and lets you use and practice what you've learned.



What this course is and isn't

➤ IS

- Study and used close-to-the-metal low-level APIs to write optimized code for GPUs
- Understand the hardware microarchitecture and the hardware-software interface ISA
- Learn to use the practical tools to design, implement, test, tune GPU compute layer, which provides the under-the-hood computation layer for higher-level applications
- For people who want to **design/develop** the GPU layers for higher-level applications, or for designing/develop GPU compute which **doesn't exist today**, or optimize to squeeze the **absolute maximum performance** from specific HW.

➤ IS NOT

- a course on higher-level applications and problems which use/benefit from GPU computing,
 - such as machine-learning/deep learning frameworks, scientific, engineering, and medical software applications, etc...
 - *Although* we will look at examples of these applications and how the GPU compute layer provides acceleration through massive parallel execution of key performance-critical algorithms.

Course Schedule*

- **Week 1 (1/6)** : Introduction, evolution, and overview of parallel GPU computing. First look at OpenCL & CUDA.
- **Week 2 (1/13)** : GPU background and microarchitecture 101. Intro to GPU programming model(s) and APIs.
 - Homework-1 DUE : 19 JAN, 11:59 PM
- **Week 3 (1/20)** : kernel languages. Profiling and HW Arch 201 and iterative optimization. Analysis of algorithms. Roofline model.
 - Homework-2 DUE : 26 JAN, 11:59 PM
- **Week 4 (1/27)** : Debugging kernels. GPU HW Arch 301. GEMM kernels.
 - Homework-3 DUE : 2 FEB, 11:59 PM
- **Week 5 (2/3)** : SIMD/SIMT. Performance analysis. Synchronization and atomics. Advanced kernel language features.
 - Homework-4 DUE : 9 FEB, 11:59 PM
- **Week 6 (2/10)** : Unified memory. Concurrency. Dynamic parallelism.
 - Final Project : *Proposals* DUE : 14 FEB, 11:59 PM
- **Week 7 (2/17)** : Numerical precision. Image processing, filters, PDE stencils.
 - Final Project : *Design* DUE : 23 FEB, 11:59 PM
- **Week 8 (2/24)** : Deep Learning case study, part 1.
- **Week 9 (3/3)** : DL case study, part 2.
- **Week 10 (3/10)** : Scientific computing on GPUs case study.
 - Final Project : *Final Report* DUE : Thursday 18 MAR, 11:59 PM

* subject to some revision as the quarter progresses to meet class needs/goals.



Prerequisites



- ▶ intermed c/c++ and dev envs & tools, necessary drivers(!), debugging
 - ▶ Visual Studio 2019 Community Edition
 - ▶ NVIDIA CUDA Toolkit
 - ▶ Intel OpenCL SDK
- ▶ basic computer architecture: microprocessors, interconnects, memory system
- ▶ Due to the class size and complexity of the new cloud-based environment, we will not be able to provide significant technical support and troubleshooting of basic setup, programming, and development tools usage
 - ▶ At the professional Master's level in UW ECE it is expected that proactive problem solving skills competency and independent/autonomous capacity is a given.

Course Logistics

Course Overview:

- There will be 10 lectures delivered via online.
- *Official Schedule***: Lectures are **Wednesday evenings 6-9:50 PM (pacific time, PST)**
- Hands-on programming exercises will be part of the weekly lecture.
- There will be 4 homework assignments which will consist of (a) readings, (b) theory, (c) coding.
- The remainder of the quarter will be focused on a final project which will be comprised of a fairly significant GPU code design and implementation utilizing and applying techniques learned in the class.
 - Parts: Project Proposal (10%), Design specification (35%), Final report and results (55%).

Grading:

- Homework (4) 50% (12.5% each)
- Final Project 50% (see *breakdown above*)

Course Materials:

- There is no required textbook for this course.

Reading materials for the course as well as a list of supplementary reading materials will be posted on the course website. All materials will be available in online electronic formats, either freely available public literature, through UW Library (www.lib.washington.edu), or on the class website (Canvas).

Course Policies

You may collaborate and discuss homework assignments and project design and implementation with your fellow classmates, professor, TA and others. However, the work you submit must be your own, and you must write your own code(s). Copying code and plagiarizing is of course not allowed.



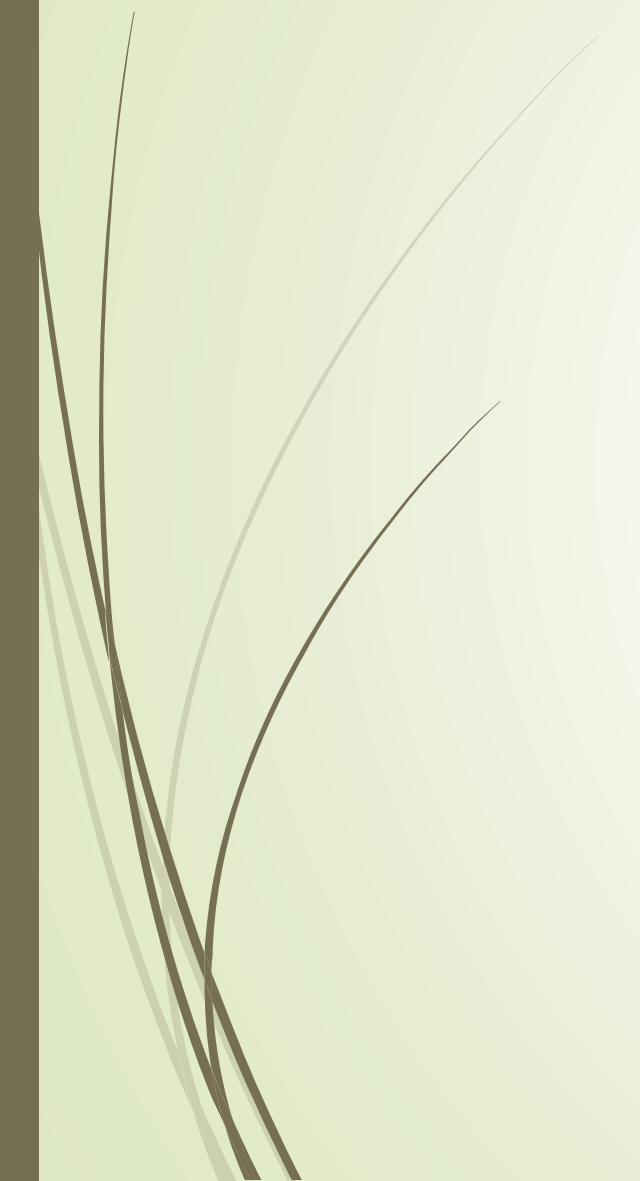
REVISED Class Schedule

Since we're no longer all sitting together once a week for 4 hrs in computer lab...

- WED evening lecture/meeting time 6-830
 - 6-7 technical lecture topic, live
 - 7-830 Individualized Q&A support, HW help, etc...
 - We'll set up some kind of request/registration form on the courseweb.
- Additional pre-recorded technical lecture segments will be posted regularly to the courseweb
 - You can watch these asynchronously on-demand on your schedule.



Course Website: Canvas

- ▶ AKA “courseweb”
 - ▶ Your primary portal to all things GPUCompute Win21
 - ▶ (Let's take a quick tour...)
- 



TA and Office Hours

Our TA: Vineetha Thomas (vthomas@uw.edu)

- Vineetha is TA'ing from India!
- Her timezone is +13.5 hours **ahead** of Seattle (PST) time.
- TA office hours can be scheduled between
 - 9AM to 12:30PM PST.
 - 6:30 PM to 10:00 PM PST
- Thus she may not be able to respond from 12:30PM to 6PM PST.
- PROFESSOR office hours
 - can be arranged on individual basis, please contact Prof R directly.



Course Reading Materials



- ▶ on courseweb in PAGES and FILES
- ▶ UW Lib website, O'Reilly books online
 - ▶ <https://www.lib.washington.edu/>



Amazon Web Services Course Cloud

- We are finalizing a brand-new setup of a cloud-based environment for the class
 - We are very lucky to receive excellent support and grants from
 - Amazon Web Services (AWS Educate)
 - NVIDIA (NVIDIA Teaching Kits)
- Cloud environment for students will include
 - virtual desktop environment (VDE) with Windows, Visual Studio 2019, CUDA Toolkit
 - any client with HTML5 browser
 - AWS EC2 GPU Instances
 - Multiple NVIDIA GPUs: Ampere A100, Volta V100, T4 Tensor GPU, M60, K80
- Details on how to access and use will be coming in next few days
 - a video guide of how to setup and access will be posted



If you have your own compatible HW, and want to use it...

--- THIS IS NOT REQUIRED ---


Compatible HW devices

- Intel graphics device (iGPU)
- NVIDIA GPU (dGPU)

Compatible SW

- Visual Studio 2019 Community Edition
- NVIDIA CUDA Toolkit 11.x
- Intel OpenCL SDK for Applications

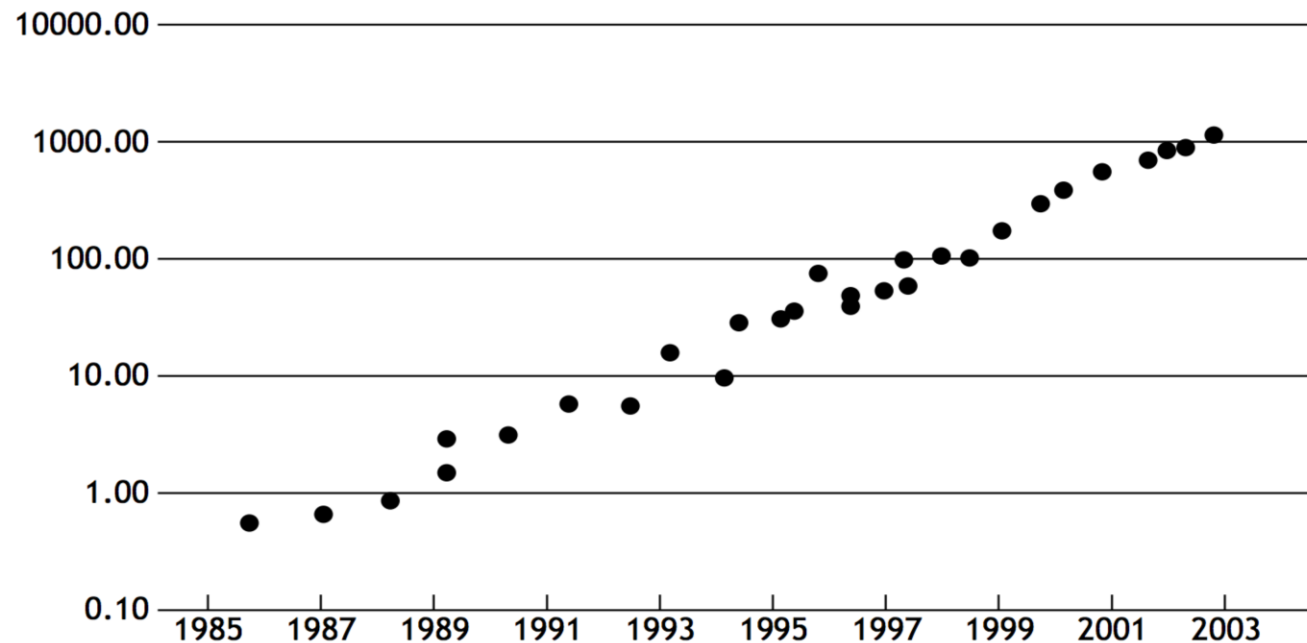
- see Exercise 1 (EX1) for more details and download links
 - also later slides at the end of Lecture 1 (L1)



Why are we studying GPUs
(and how to effectively use them)
today??

Setting Some Context

- Before we continue our multiprocessor story, let's pause to consider:
 - Q: what had been happening with single-processor performance?
- A: since forever, they had been getting **exponentially faster**
 - Why?



A Brief History of Processor Performance

■ Wider data paths

- 4 bit → 8 bit → 16 bit → 32 bit → 64 bit

■ More efficient pipelining

- e.g., 3.5 Cycles Per Instruction (CPI) → 1.1 CPI

■ Exploiting instruction-level parallelism (ILP)

- “superscalar” processing: e.g., issue up to 4 instructions/cycle

■ Faster clock rates

- e.g., 10 MHz → 200 MHz → 3 GHz

■ During the 80s and 90s: large exponential performance gains

- and then...



Hardware Evolution

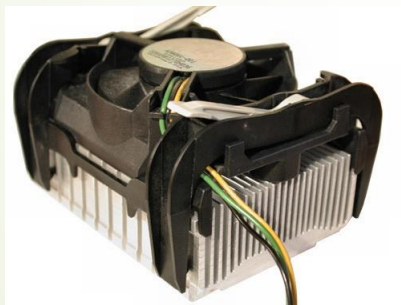
There are limits to “automatic” improvement of scalar performance:

- 1. The Power Wall:** Clock frequency cannot be increased without exceeding air cooling.
- 2. The Memory Wall:** Access to data is a limiting factor.
- 3. The ILP Wall:** All the existing instruction-level parallelism (ILP) is already being used.

What has happened in the last several years?

- ▶ Processing chip manufacturers increased processor performance by increasing CPU clock frequency
 - ▶ Riding Moore's law
- ▶ Until the chips got too hot!
 - ▶ Greater clock frequency \Rightarrow greater electrical power \Rightarrow heat!
 - ▶ Pentium 4 heat sink

○ Frying an egg on a Pentium 4



- ▶ Add multiple cores to add performance
 - ▶ Keep clock frequency same or *reduced*
 - ▶ Keep a lid on *power requirements*

Power Density Growth

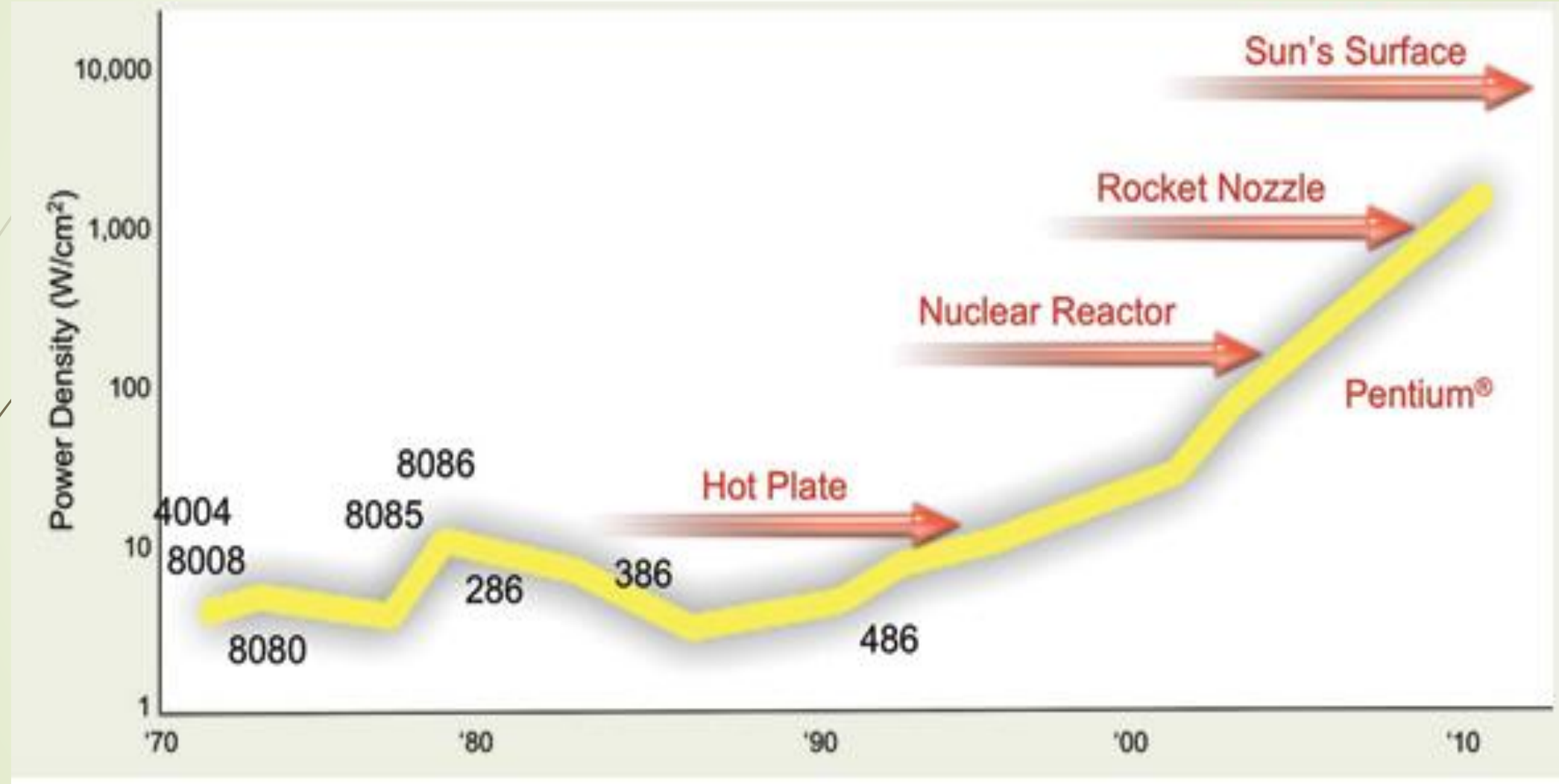
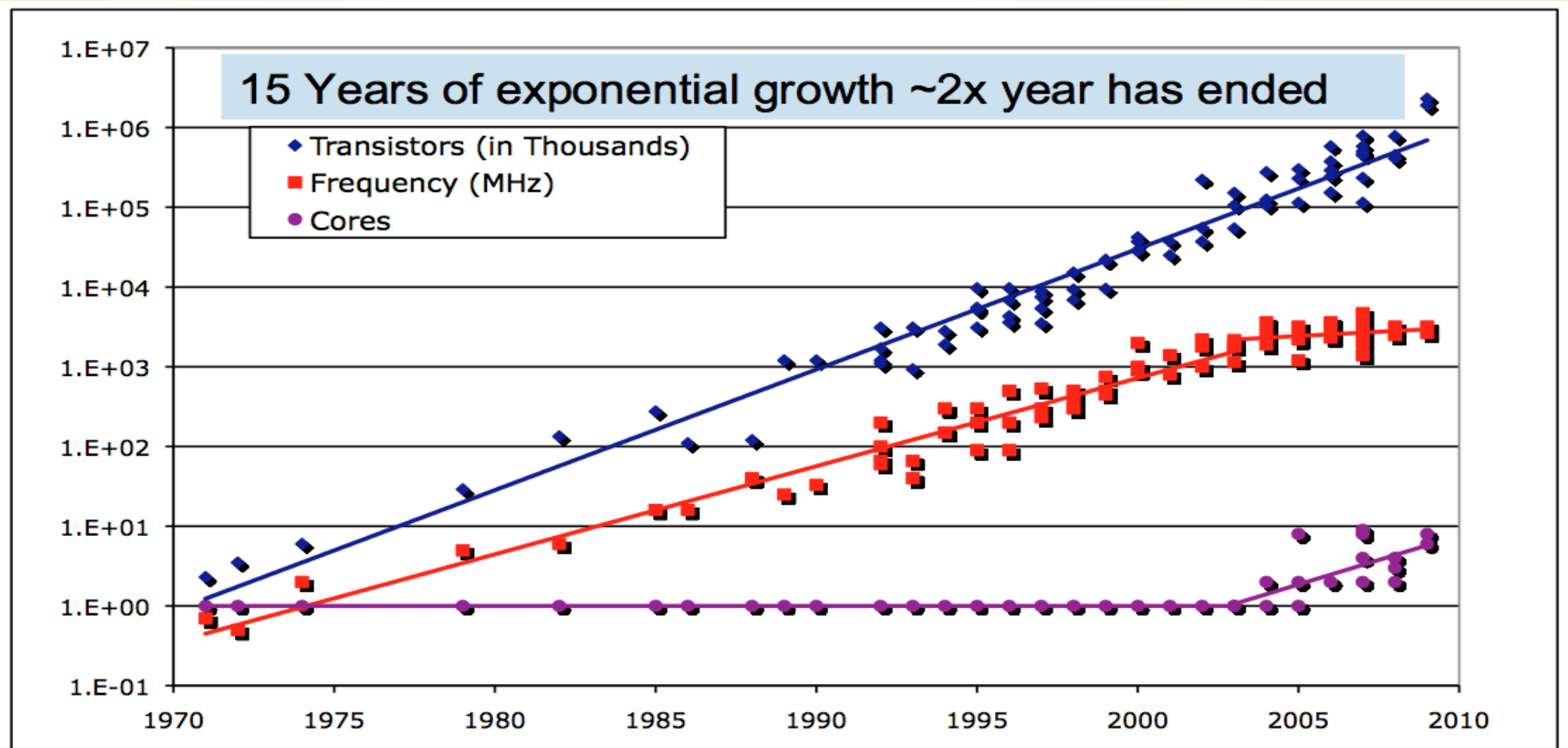


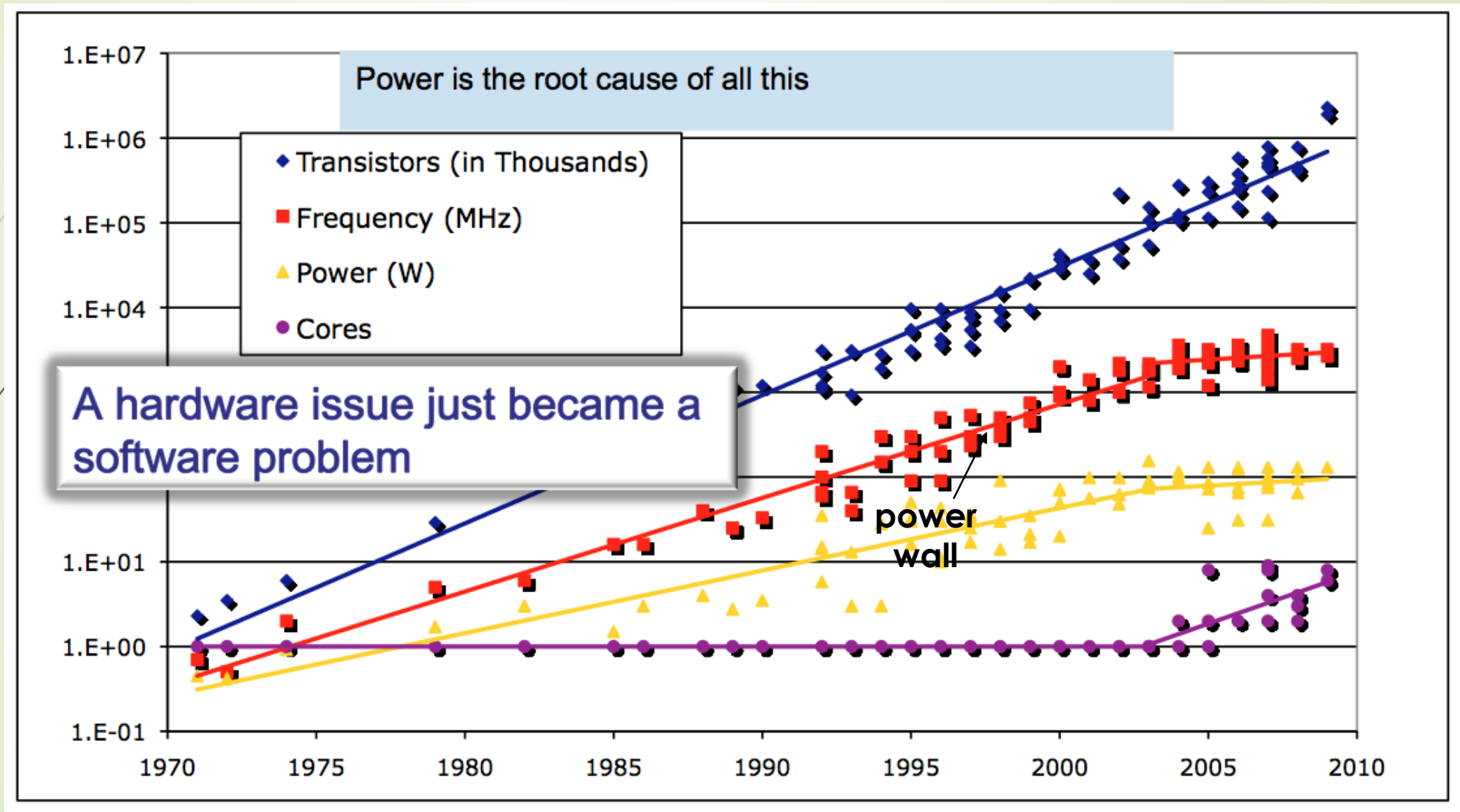
Figure courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004

What's Driving Parallel Computing Architecture?

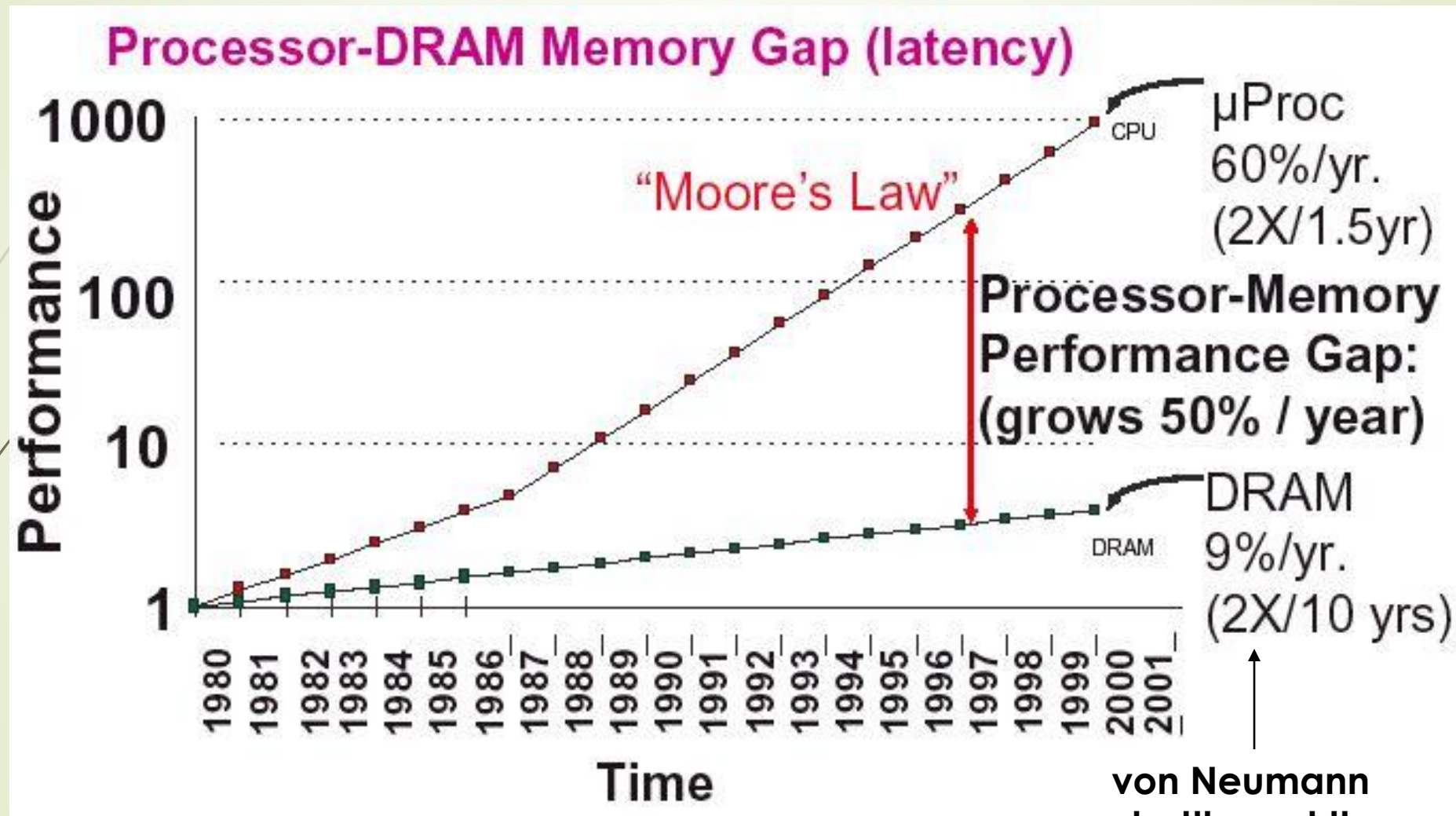


Data from Kunle Olukotun, Lance Hammond, Herb Sutter,
Burton Smith, Chris Batten, and Krste Asanović
Slide from Kathy Yelick

What's Driving Parallel Computing Architecture?

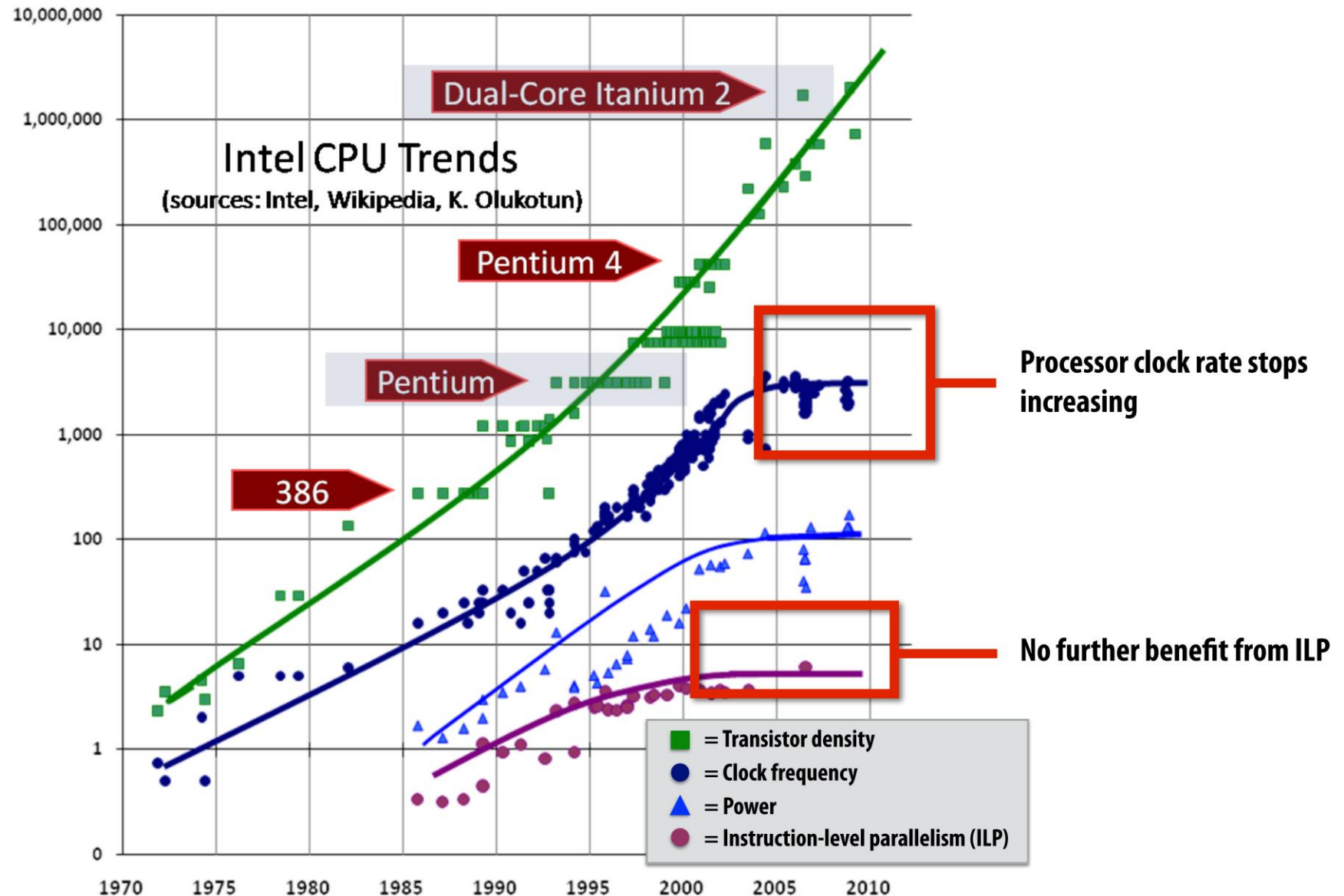


What's Driving Parallel Computing Architecture?

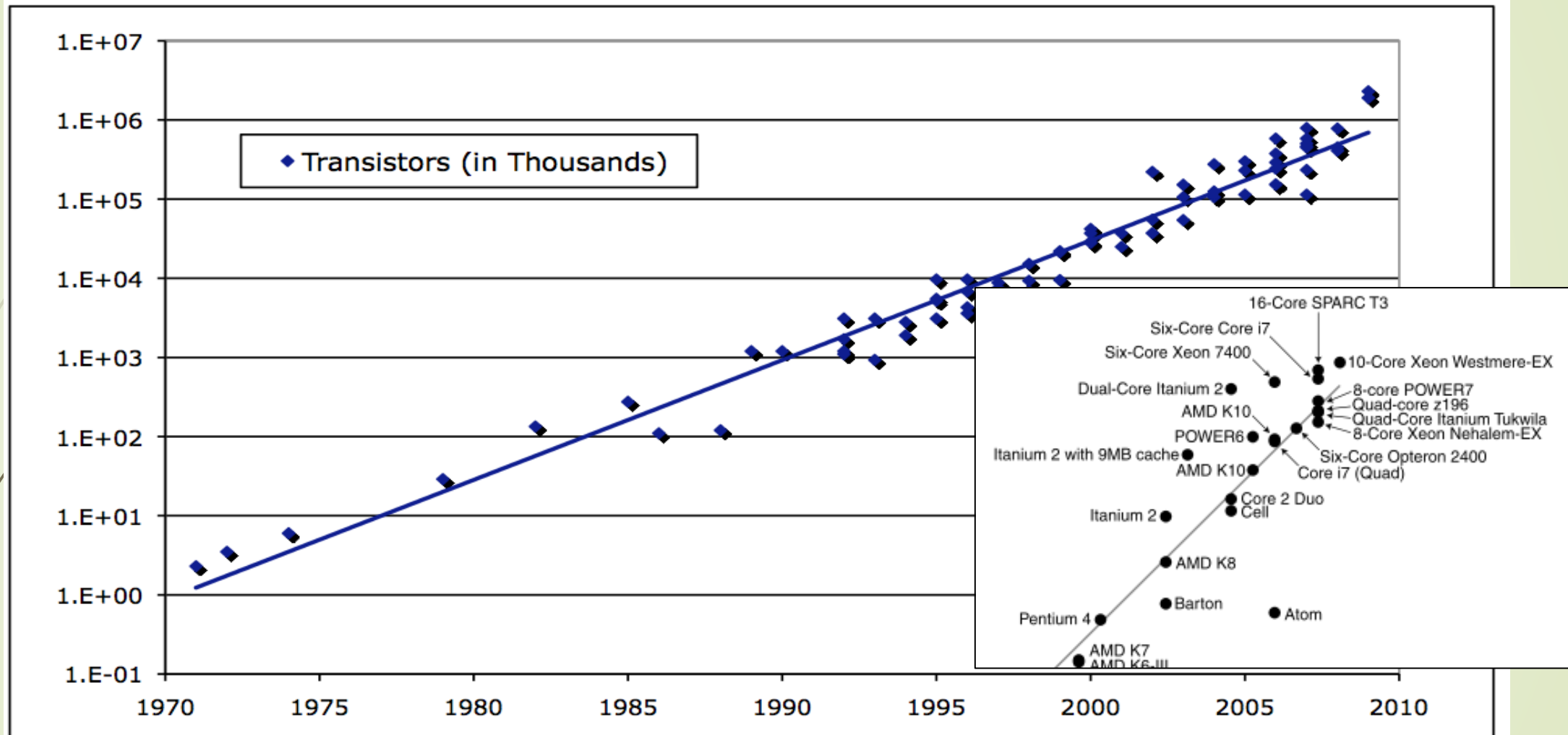


von Neumann
bottleneck!!
(memory wall)

ILP tapped out + end of frequency scaling

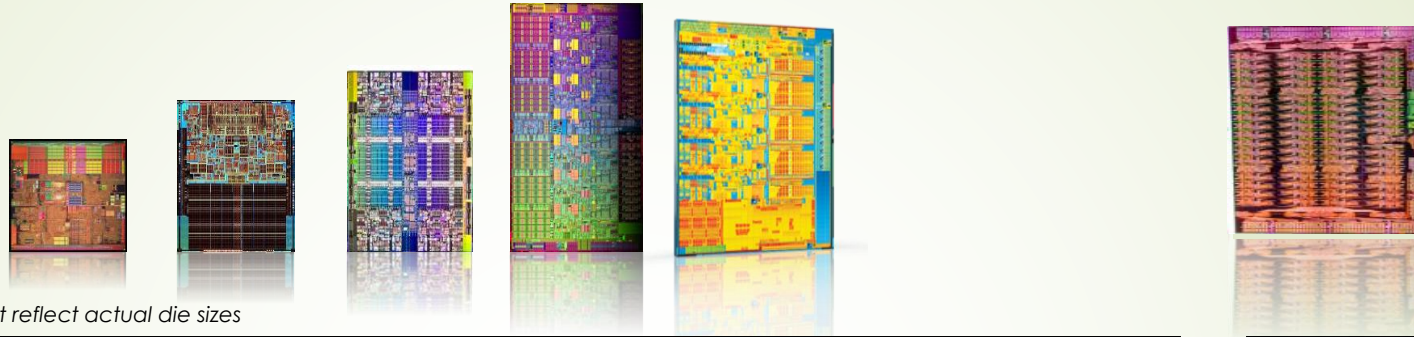


Microprocessor Transistor Counts (1971-2011)



Data from Kunle Olukotun, Lance Hammond, Herb Sutter,
Burton Smith, Chris Batten, and Krste Asanović
Slide from Kathy Yelick

Trends: More cores. Wider vectors. Coprocessors.



Intel® Xeon®
processor
64-bit

Intel® Xeon®
processor
5100 series

Intel® Xeon®
processor
5500 series

Intel® Xeon®
processor
5600 series

Intel® Xeon®
processor
code-
named
Sandy
Bridge

Intel® Xeon®
processor
code-
named
Ivy Bridge

Intel® Xeon®
processor
code-
named
Haswell

Intel® Xeon
Phi™
coprocessor
code-named
Knights
Corner

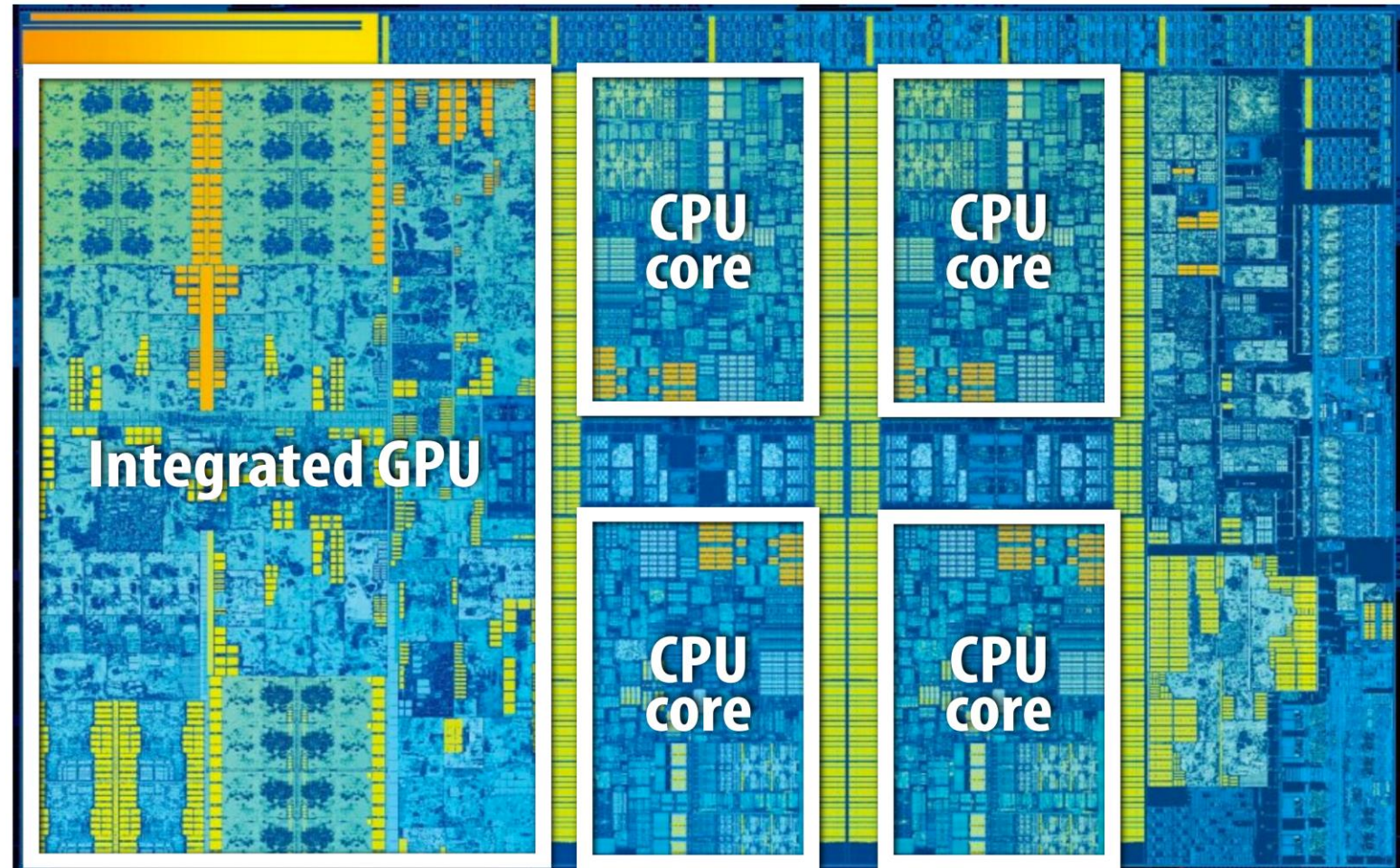
Core(s)	1	2	4	6	8			57-61
Threads	2	2	8	12	16			228-244
SIMD Width	128	128	128	128	256	256	256	512
	SSE2	SSSE3	SSE4.2	SSE4.2	AVX	AVX	AVX2 FMA3	IMCI

Software challenge: Develop scalable software

system-on-chip
(SOC)

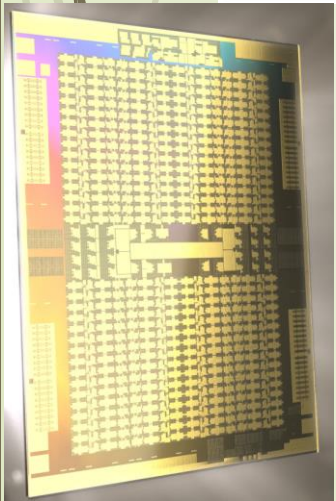
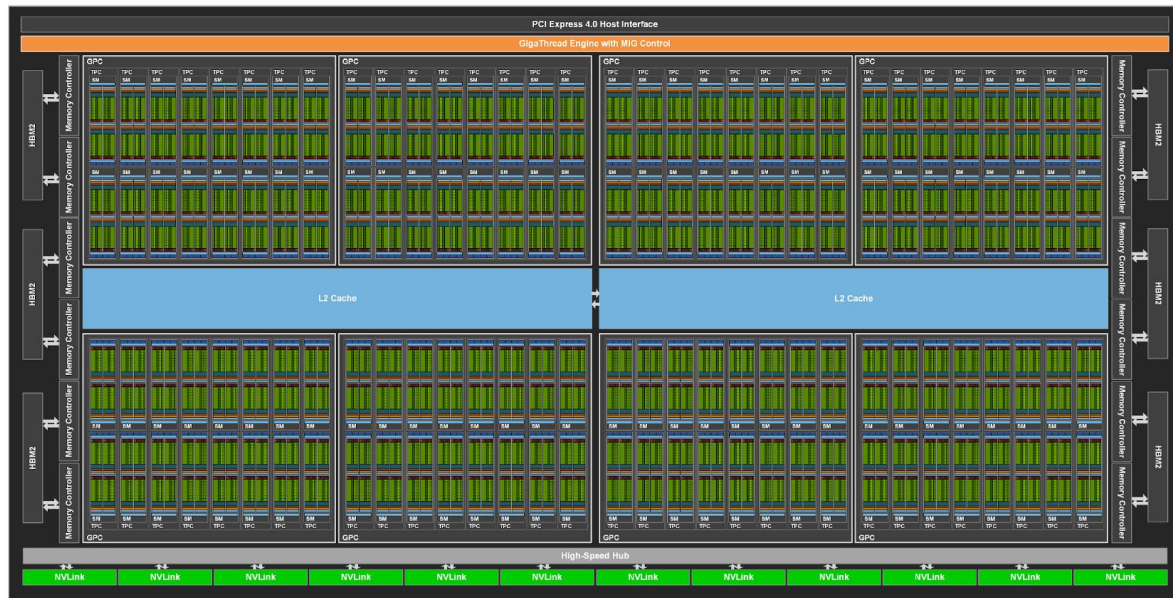
Intel Skylake (2015) (aka "6th generation Core i7")

Quad-core CPU + multi-core GPU integrated on one chip



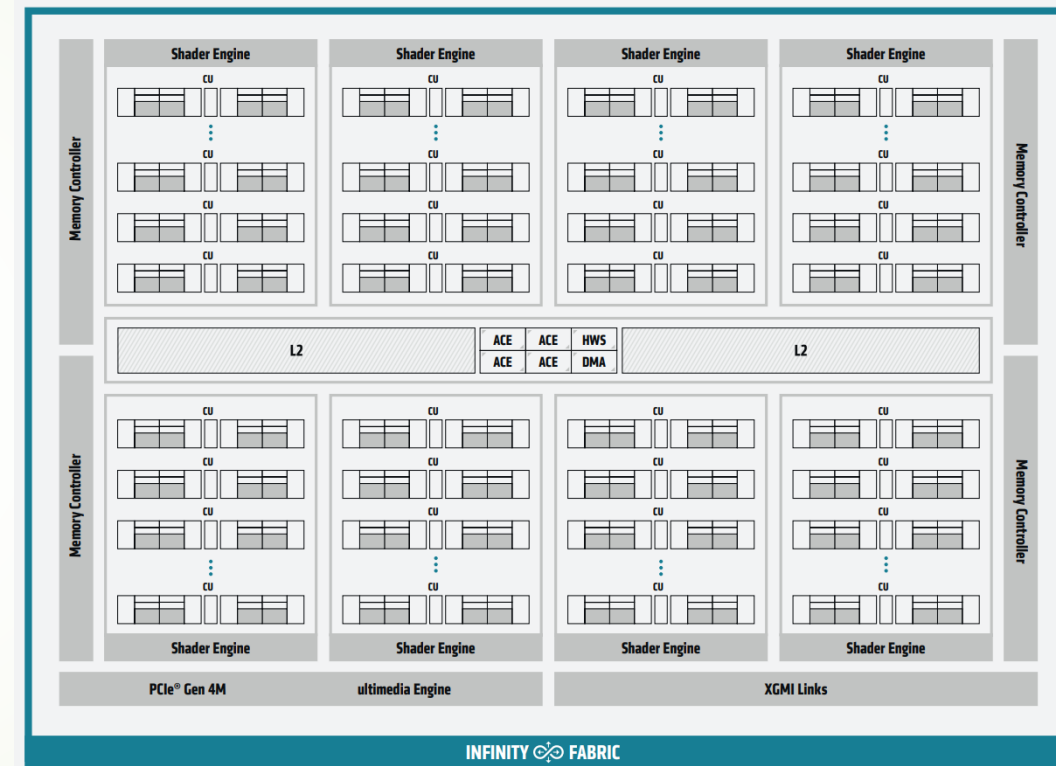
Discrete GPUs

NVIDIA GA100
Ampere architecture



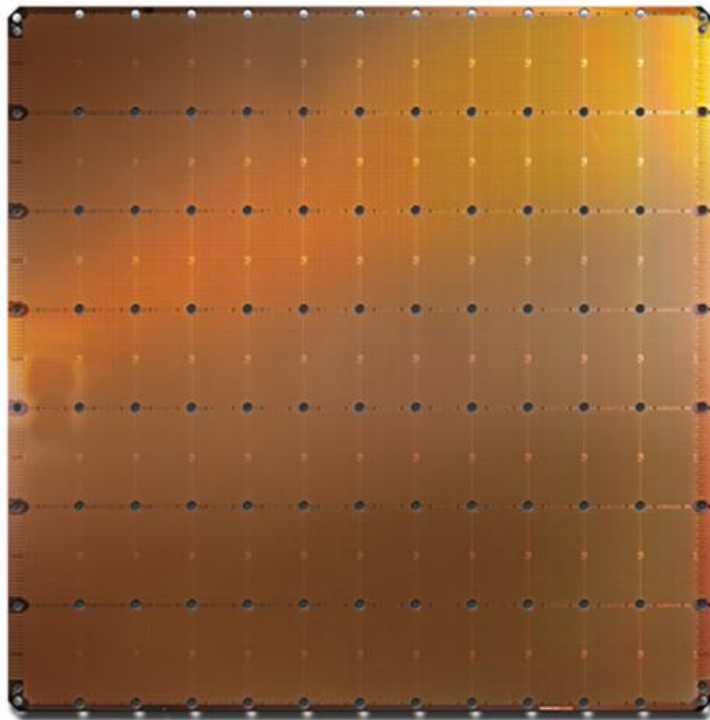
Intel Iris X^e Max
“Tiger Lake” SOC : X^e architecture (w 11th gen Intel Core)
first Intel entry in discrete GPU market
(not a lot of details available yet...)

AMD Instinct MI100
CDNA architecture



What's next???

➤ Cerebras Wafer-Scale Engine (WSE)??



Cerebras WSE
1.2 Trillion transistors
46,225 mm² silicon



Largest GPU
21.1 Billion transistors
815 mm² silicon

	Cerebras WSE	Largest GPU	Cerebras Advantage
Chip size	46,225 mm ²	815 mm ²	56.7 X
Cores	400,000	5,120	78 X
On chip memory	18 Gigabytes	6 Megabytes	3,000 X
Memory bandwidth	9 Petabytes/S	900 Gigabytes/S	10,000 X
Fabric bandwidth	100 Petabits/S	300 Gigabits/S	33,000 X



Terms

- ▶ **multi-core**: CPUs (10s of complex cores)
- ▶ **many-core**: GPU (100s, 1000s of simpler cores)
- ▶ **heterogeneous** systems/computing:
 - ▶ using both CPU + GPU processing resources

Summary: Parallelism and Performance

There are limits to “automatic” improvement of scalar performance:

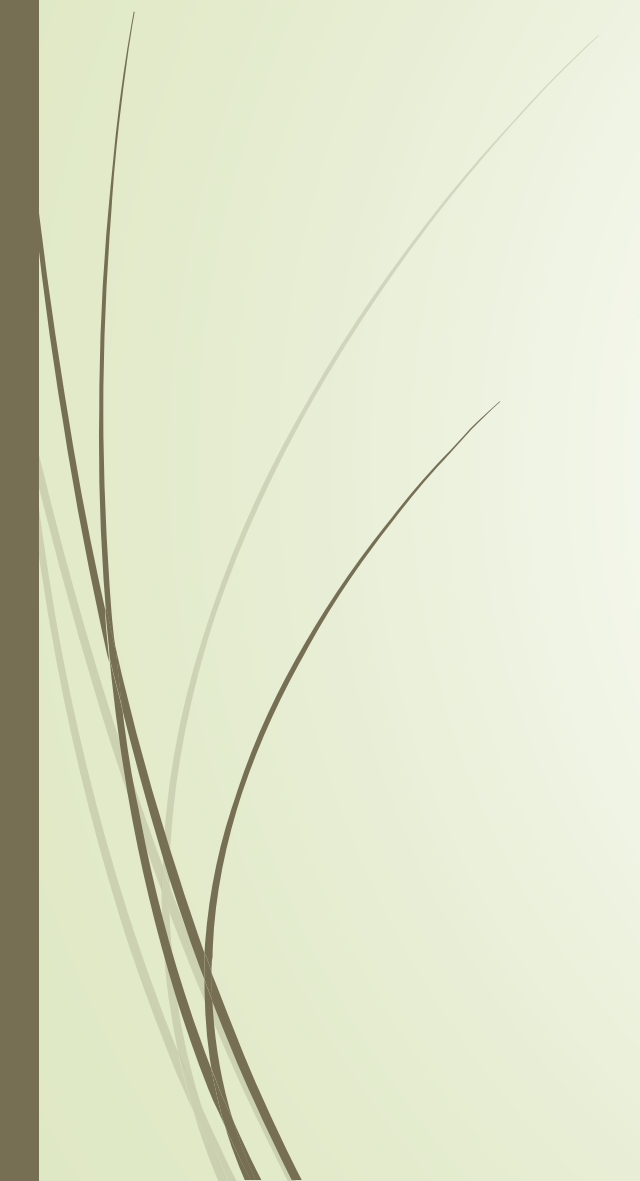
1. **The Power Wall:** Clock frequency cannot be increased without exceeding air cooling.
2. **The Memory Wall:** Access to data is a limiting factor.
3. **The ILP Wall:** All the existing instruction-level parallelism (ILP) is already being used.

→ **Conclusion:** *Explicit* parallel mechanisms and *explicit* parallel programming are *required* for performance scaling.

→ require specialized knowledge and skills, which we'll develop in this course



3 Goals of Parallel Computing

1. Solve a given problem in less time.
 2. Solve a bigger problem in same (given) amount of time.
 3. Achieve better solutions for given problem in given amount of time.
 1. More complex, more accurate models
- 



Good parallelism candidates

- Large problem sizes
- High modeling complexity
- Examples:
 - HPC (High performance computing)
 - physics, chemistry, genetics, biomedical, engineering
 - Machine learning (ML/DL)
 - big data, analytics



Concept of Parallel Execution on GPUs



Time versus Space

- ▶ Counting pennies
 - ▶ 1 person, N hours
 - ▶ N people, $N/N = 1$ hour !?
 - ▶ But: need bigger room, more chairs, desks, etc.
- ▶ Tradeoff: Time vs Space
 - ▶ time: serial, sequential execution
 - ▶ space: parallel, concurrent execution
- ▶ GPUs preference space
- ▶ But how, when can this provide benefits?

Example 1: vector addition

- Add two vectors (**A,B**) of length N

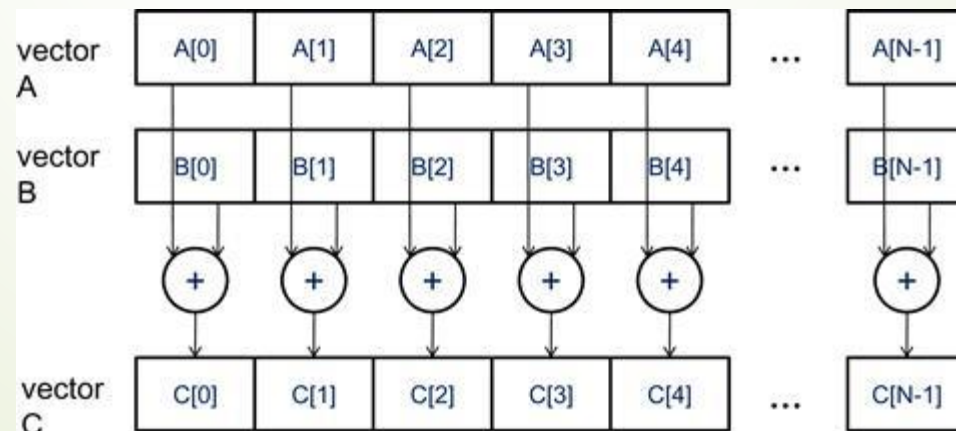
$$\mathbf{C} = \mathbf{A} + \mathbf{B}$$

- Sequential (serial) code implementation = for loop

```
for(int x = 0; x < N; x++)
```

```
    C[x] = A[x] + B[x];
```

- **KEY:** Addition of individual vector elements is independent
- Allows data parallelism



Vector Addition – Serial C Implementation

```
void vecadd_idx(int n, const float *a, const float *b, float *c)
{
    int x = 0;

    for (x; x < n; x++)
        c[x] = a[x] + b[x];
}
```

```
//=== ALTERNATE (pointer-based) ===
void vecadd_ptr(int n, const float *a, const float *b, float *c)
{
    int x = 0;

    for (x; x < n; x++, a++, b++, c++)
        *c = *a + *b;
}
```


Loop Unrolling

```
for (int x = 0; x < 8; ++x) {  
    C[x] = A[x] + B[x];  
}
```

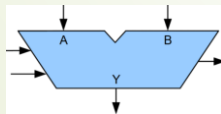
sequential, single core

time

↓

$C[0] = A[0] + B[0]$
 $C[1] = A[1] + B[1]$
 $C[2] = A[2] + B[2]$
 $C[3] = A[3] + B[3]$
 $C[4] = A[4] + B[4]$
 $C[5] = A[5] + B[5]$
 $C[6] = A[6] + B[6]$
 $C[7] = A[7] + B[7]$

repeat
N times



Instead what if we had multiple cores?

Due to the **independence** of each element-wise sum, we can execute in parallel.

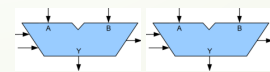
{2-wide (2 lanes)}

$A[0] + B[0]$ $A[1] + B[1]$

$A[2] + B[2]$ $A[3] + B[3]$

...

$A[6] + B[6]$ $A[7] + B[7]$

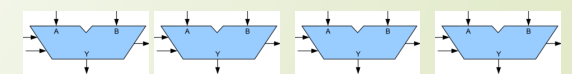


OR

{4-wide}

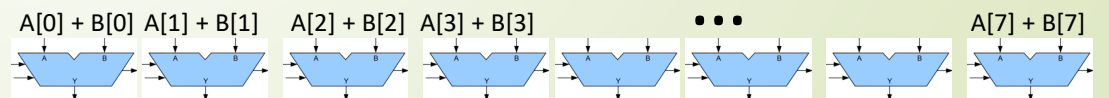
$A[0] + B[0]$ $A[1] + B[1]$ $A[2] + B[2]$ $A[3] + B[3]$

$A[4] + B[4]$ $A[5] + B[5]$ $A[6] + B[6]$ $A[7] + B[7]$



OR

{8-wide}



Computational “Index Space”

In a 1D problem there is just one index: $\{x\}$, $\{i\}$, $\{u\}$, or ...

A_0	A_1	A_2	A_3
-------	-------	-------	-------

in 2D we need 2 indices : $\{x,y\}$, $\{i,j\}$, $\{u,v\}$ or ...
(and 3 for 3D: $\{x,y,z\}$, $\{i,j,k\}$, $\{u,v,w\}$, or ...)

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}
A_{30}	A_{31}	A_{32}	A_{33}

we can sub-divide the full *grid* into a partition of smaller *blocks* or *groups*

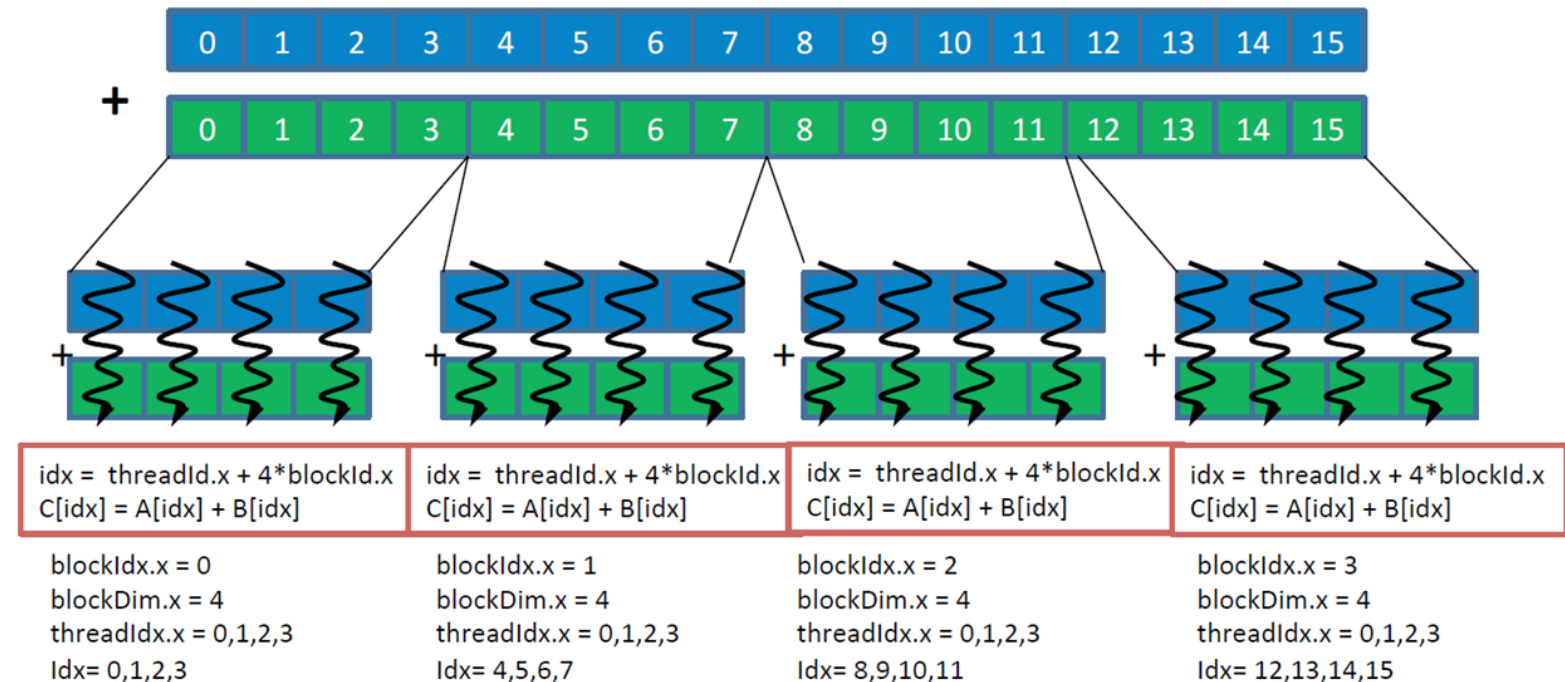
Consider
1D vector addition

$$\mathbf{C} = \mathbf{A} + \mathbf{B}$$

where vectors are 16
elements long.

We sub-divide into
4 *blocks* (*work-groups*)
of 4 *threads* (*work-items*)
each.

Terms are CUDA (OpenCL)



Vector Addition – GPU Kernel (Device-side)

Both OpenCL and CUDA kernel languages use extensions and subset of standard C. Slight differences in syntax, but very similar.

```
// CUDA kernel (*.cu)
```

```
__global__ void vecAdd1D( float *a, float *b, float *c)
{
    int gid = threadIdx.x + blockDim.x * blockIdx.x;
    c[gid] = a[gid] + b[gid];
}
```

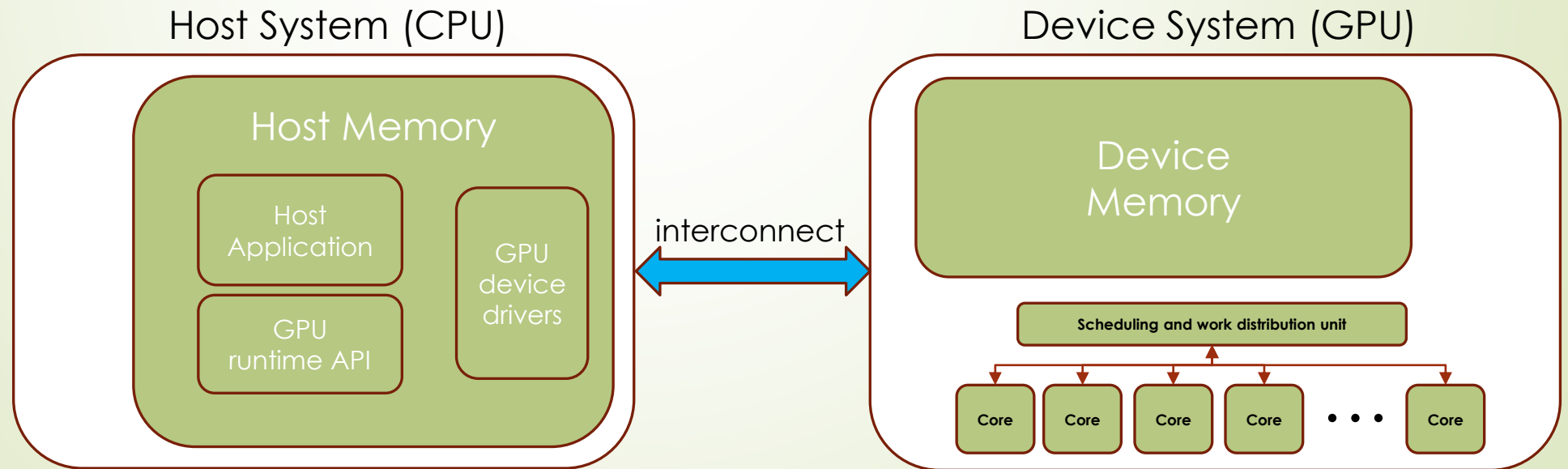
```
// OpenCL kernel (*.cl)
```

```
__kernel void vecAdd1D( __global const float *a, __global const float *b, __global float *c)
{
    int gid0 = get_global_linear_id();
    // int gid0 = get_local_id(0) + get_local_size(0) * get_block_id(0); // equivalent
    c[gid0] = a[gid0] + b[gid0];
}
```

How does this all work?

- ▶ **Host** application
 - ▶ GPU API (e.g. CUDA or OpenCL API)
 - ▶ kernel configuration and dispatch
 - ▶ memory management and data transfer
 - ▶ GPU device drivers : vendor-specific (and model specific...)
- ▶ **Device** kernel functions
 - ▶ execute computational work

Simplified System Diagram





Application Development Overview

- #include appropriate API header files
- implement host application using GPU API
 - NVIDIA CUDA
 - Runtime API – *simple to use but limited*
 - Driver API – *more complex but full control at low-level*
 - Khronos OpenCL
 - host API – *complex but full control at low-level*
 - *with vendor-specific driver layer*
- write kernel function(s)
 - using CUDA C++ kernel extensions
 - or OpenCL C language
- compile and link to runtime libraries (static or dynamically)



Application Dev Overview



- ▶ In this class, we want to work as closely to HW as possible
 - ▶ we'll primary use the low-level GPU C/C++ APIs to get "close to metal"
 - ▶ CUDA Driver API
 - ▶ OpenCL API
 - ▶ At some points we'll look deeper
 - ▶ Intermediate Representation assembly language (PTX, GEN)
 - ▶ Device-specific assembly language (SASS, HCN)
- ▶ For our very first GPU exercise/HW, we'll use the simpler CUDA Runtime API
 - ▶ it simplifies/eliminates many of the steps...

Basic Host Application Template

This is a general guideline for the host application structure.

- Not all steps are always used, and sometimes the order may be changed.
- Also for more advanced applications, additional steps may be added.

```
int main( ) {  
    // define/init static/const variables  
    // initialize GPU host API  
    // query for platform/device information  
    // setup GPU host API environment and device program(s)  
    // allocate host memory variables h_  
    // initialize host memory vars  
    // allocate device memory vars  
    // set up kernel arguments on device  
    // copy host memory to device memory  
    // determine GPU device kernel execution configuration  
    // launch kernel on device  
    // wait for kernel execution to complete, check for errors  
    // retrieve results from device  
    // use/check results  
}
```

CUDA Runtime API

- In the simplest case, you can just have one source file <srcname>.cu
- The file will contain

```
// define your CUDA kernel here  
  
int main( ) {  
    // host application code using CUDA runtime API calls here  
}
```

- Compile and run the file using NVCC compiler

```
nvcc -o <srcname> <path-to-src>/<srcname>.cu -run
```

CUDA Runtime API

- ▶ We discussed the conceptual “index space” (computation domain)
 - ▶ and how to access each kernel instance’s unique index values
 - ▶ `threadIdx.x`, `blockIdx.x`, etc...
- ▶ Now let’s see how to define it in practice to launch/dispatch a GPU computation
- ▶ For the CUDA Runtime API we use
 - ▶ The kernel *execution configuration* syntax `<<<...>>>`
 - ▶ (syntax will differ somewhat for Driver API and OpenCL but basic concepts same)

- ▶ Consider a CUDA GPU kernel named `cudaFoo`

```
__global__ void cudaFoo(int a) {  
    int myIdx = threadIdx.x + blockDim.x * blockIdx.x; // for 1D index space  
    // do kernel calcs  
}
```

- ▶ then in host appl

```
int threadsPerBlock = 4*32; // on CUDA devices, warps contain 32 threads  
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;  
cudaFoo<<< blocksPerGrid, threadsPerBlock>>>>(d_a);
```

GPU API Error Handling concepts

- In GPU applications, errors can occur *synchronously* or *asynchronously*
 - calls made from the host to the device can occur *asynchronously*
 - the call will return immediately, before the function has actually completed on GPU
- CUDA Runtime API
 - `cudaGetErrorName`
 - `cudaGetErrorString`
 - `cudaGetLastError`
 - `cudaPeekAtLastError`
- CUDA Driver API
 - function return values
 - `cuGetErrorName`
 - `cuGetErrorString`
- OpenCL API
 - function return values (synchronous)
 - event model (*we'll cover this fully later on.*)
 - `clWaitForEvents()`
 - `clGetEventInfo()`

CUDA Runtime API Error Handling

CUDA kernels return `void` not `cudaError_t` so the error handling macro can't be used for checking kernel launch errors

```
vectorAddKernel<<<numBlocks,numThreads>>>>(c,b,a,N);
cudaError_t err;
err = cudaGetLastError(); // `cudaGetLastError` will return the error from above.
if (err != cudaSuccess)
{
    printf("Error: %s\n", cudaGetErrorString(err));
}
```

// Error Handling Macro

```
#include <stdio.h>
#include <assert.h>
```

```
inline cudaError_t checkCuda(cudaError_t result)
{
    if (result != cudaSuccess) {
        fprintf(stderr, "CUDA Runtime Error: %s\n", cudaGetErrorString(result));
        assert(result == cudaSuccess);
    }
    return result;
}
```

```
int main()
{
    // The macro can be wrapped around any function returning a value of type `cudaError_t`.
    checkCuda( cudaDeviceSynchronize() )
}
```



SW Development Environment(s)

- Microsoft Visual Studio 2019 Community edition (FREE)
 - Only Dev IDE supported for the class!
 - Windows OS only
 - Install <https://visualstudio.microsoft.com/downloads/>
 - DOCS <https://docs.microsoft.com/en-us/visualstudio/windows/?view=vs-2019>
- Eclipse
 - NVIDIA tools support
 - Linux version
 - *I don't know this tool deeply so I won't be able to provide help with it!*

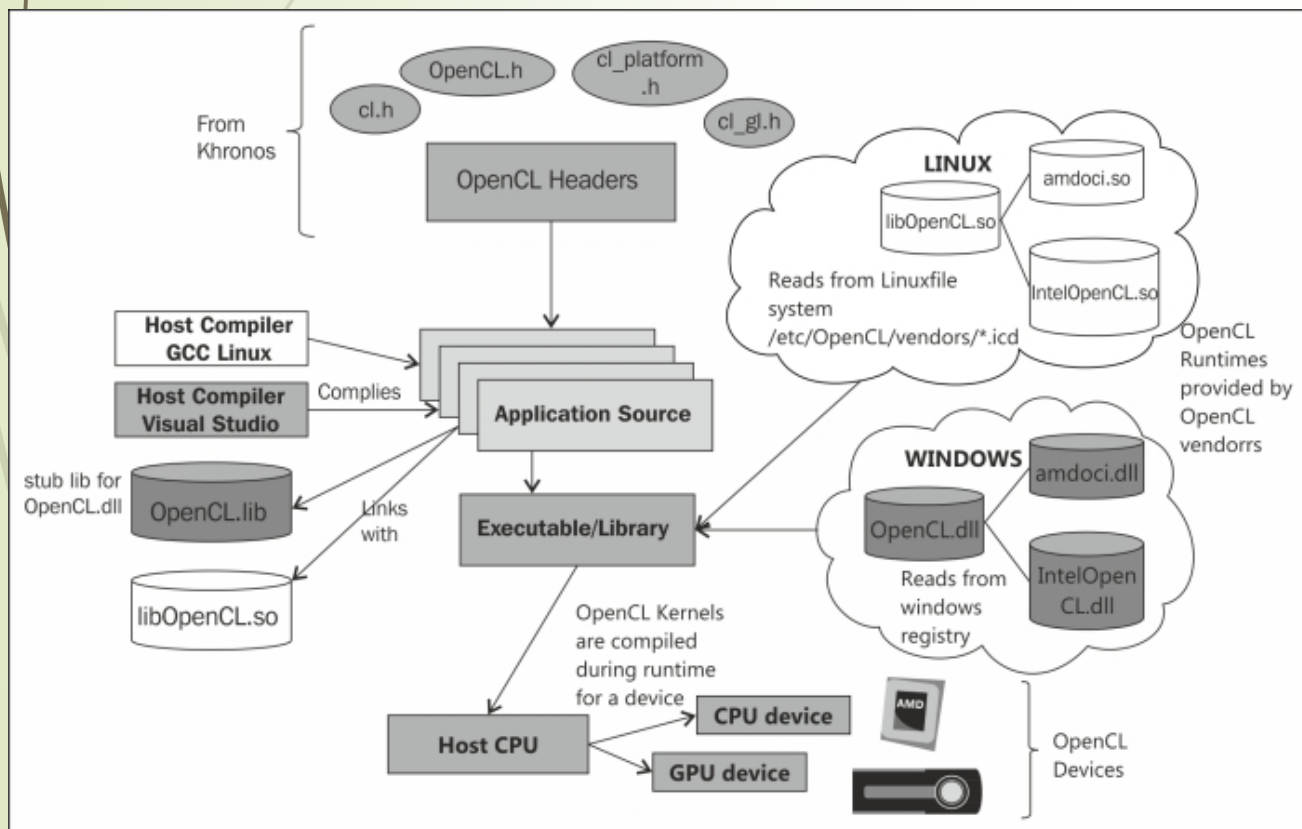
GPU SDKs

- NVIDIA CUDA Toolkit – *Primary for this class*
 - v11.x <https://developer.nvidia.com/cuda-downloads>
 - provides support for both CUDA (runtime & driver APIs) and OpenCL 1.2
 - DOCS <https://docs.nvidia.com/cuda/index.html>
- Intel
 - SDK for OpenCL Applications
 - <https://software.intel.com/content/www/us/en/develop/tools/opencl-sdk.html>
 - Intel Compute Runtime for Linux
 - <https://01.org/compute-runtime>
 - <https://github.com/intel/compute-runtime>
- AMD
 - ROCm
 - <https://github.com/RadeonOpenCompute>

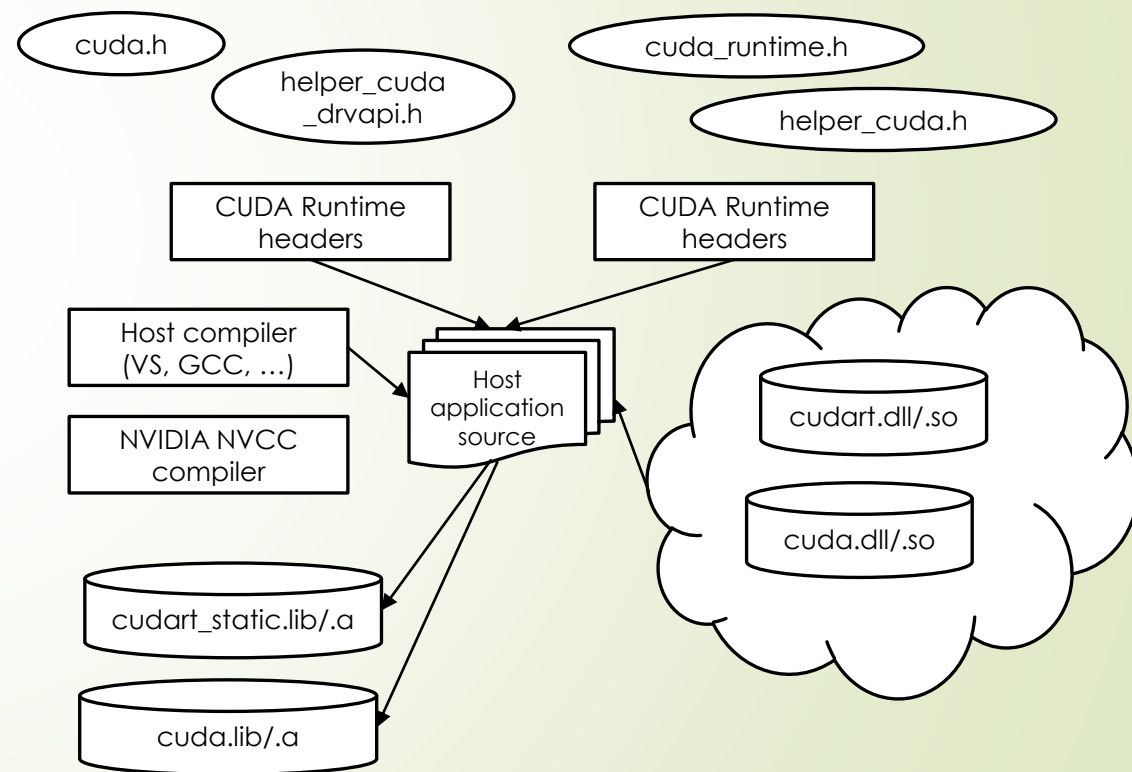
GPU Compute Environment Components

OpenCL

Windows OpenCL ICD loader (available OpenCL platforms):
HKEY_LOCAL_MACHINE\SOFTWARE\Khronos\OpenCL\Vendors



CUDA



All CUDA support from NVIDIA CUDA Toolkit



Exercise Set #1

DUE: (ideally before 13 WED L2, so you can prepare questions)

See: <Canvas Classweb>/Files/Homeworks/EX1

IMPORTANT NOTE: Although there is no official homework due for this EX1 set, it is **critical** you work through it carefully and thoroughly, as it will provide the foundation for *everything* we do in this class. Without facility in these basic skills you will surely struggle to succeed in this class.

- it will also be a chance for you to self-assess that you have the necessary prerequisites