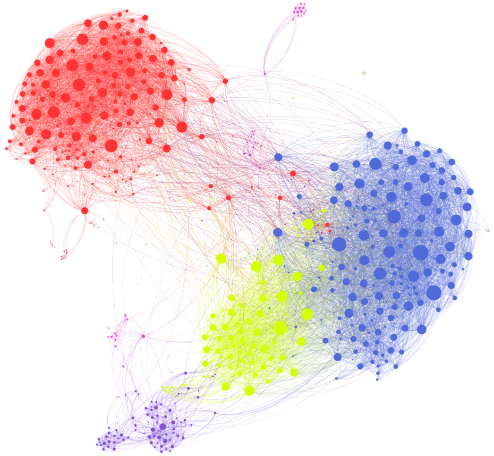
Leveraging the Power of GPUs

An Introduction to High Performance Computing

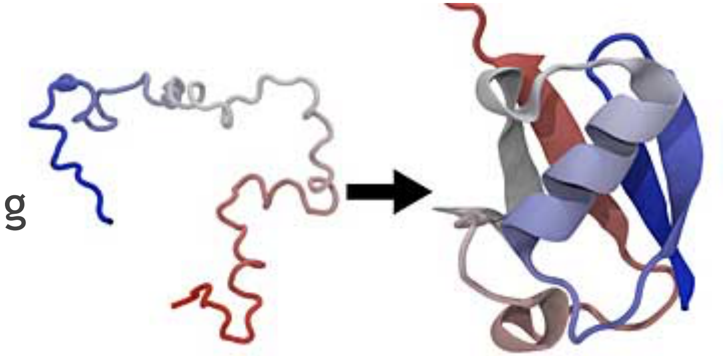
William Killian

What is High Performance Computing?

- **Using fast, parallel systems to solve:**
- Complex problems
  - Social network interactions



- Large problems
  - Protein folding

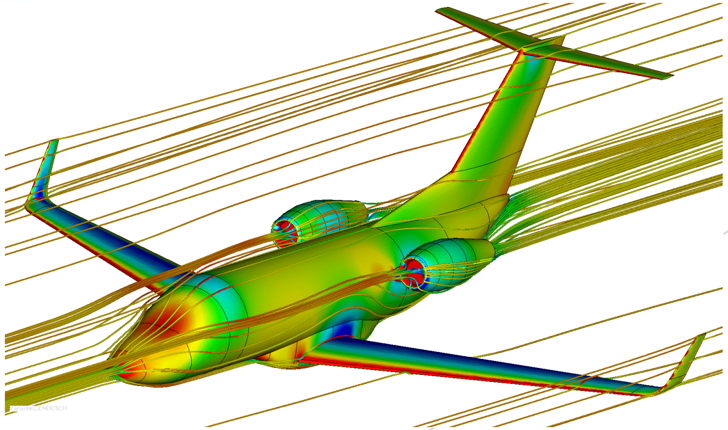


- Compute-intensive problems
  - Physics simulations (fluid dynamics)

Amazon Web Services (aws.amazon.com/hpc)

Components:

- Network
- Storage
- Memory
- Compute



Motivation for Parallelism

- We have traditionally programmed on single-core architectures
- Still taught predominantly about sequential programming
  - Imperative, *iterative*, **stateful** programming languages: Java, C++, C#
  - Parallelism is an afterthought
- With some exceptions:
  - Web Programming: AJAX, responsive/reactive loading
  - Computer Architecture: Instruction-level parallelism, instruction ordering
  - Operating Systems: processes, threads, mutexes
  - Networks: asynchronous data transfer, out-of-order packet analysis

A (Brief) History of Parallel Architectures

1. Sequential Core

- Single Instruction
- Single Data Element

2. Pipelined Core

- Single Instruction
- Multiple Instructions "In Flight"

3. Vector Machine

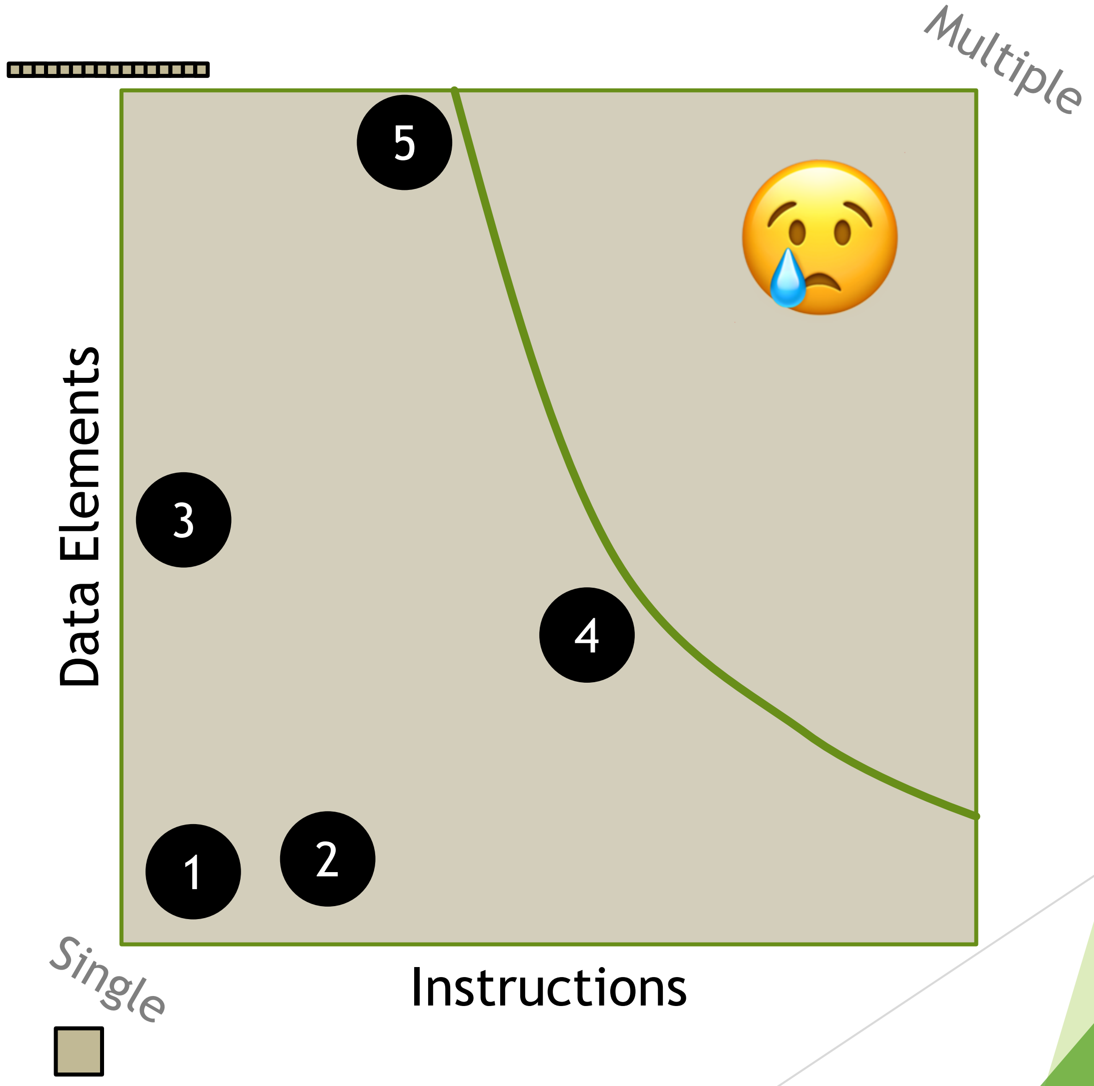
- Single Instruction
- Multiple Data Elements

4. Multi-core

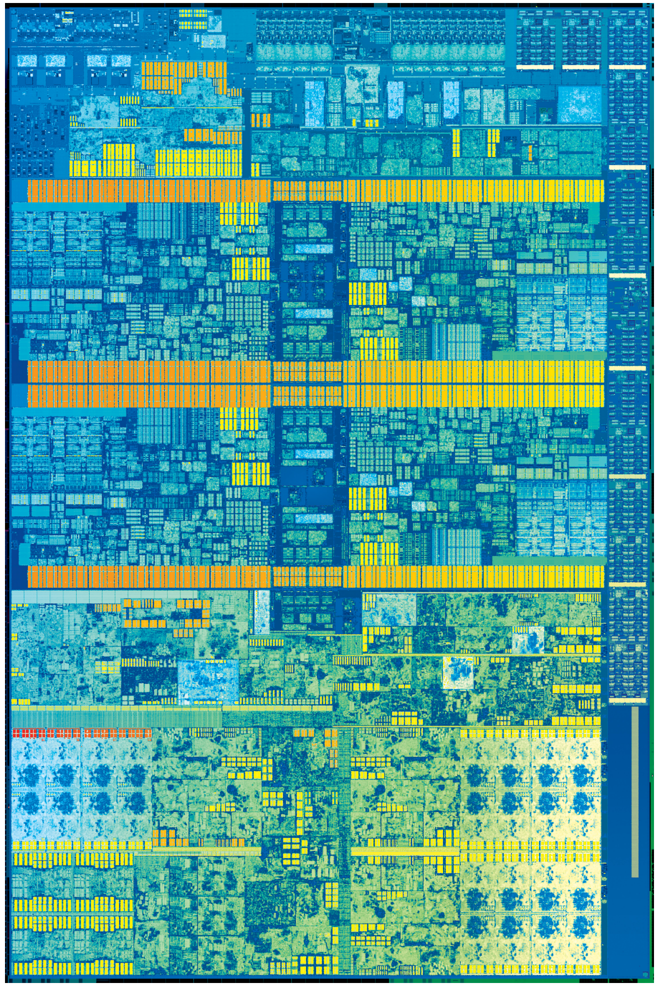
- Many Instructions
- Multiple Data Elements

5. GPUs

- Single Program
- Multiple Data Elements

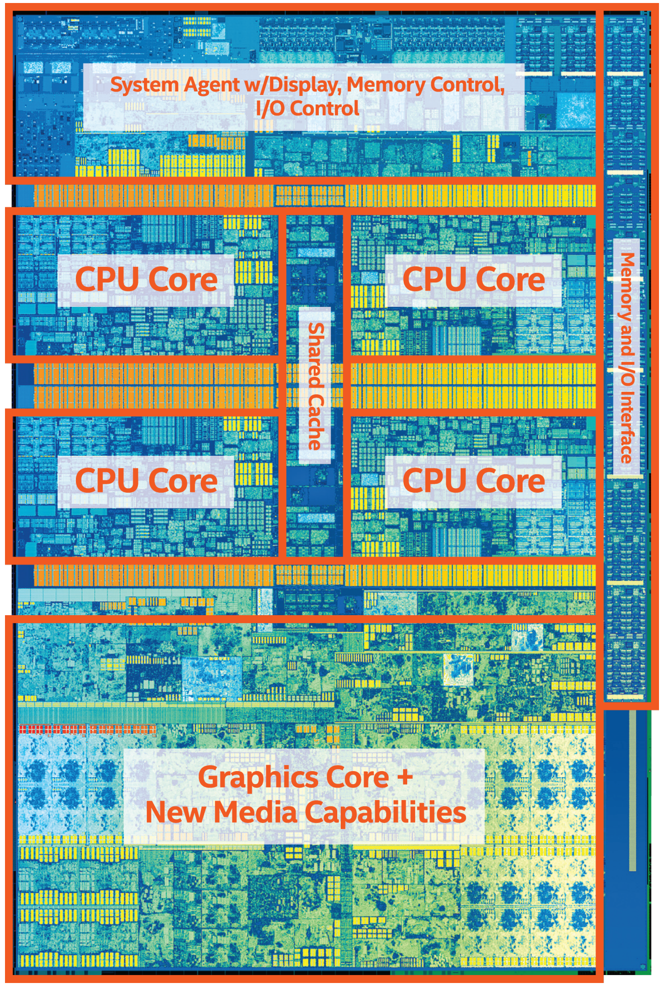


A Modern CPU (Intel Core i7-7700K)



- Die Layout?
- How much is:
  - Compute?
  - Graphics?
  - System I/O
  - Memory I/O

A Modern CPU (Intel Core i7-7700K)

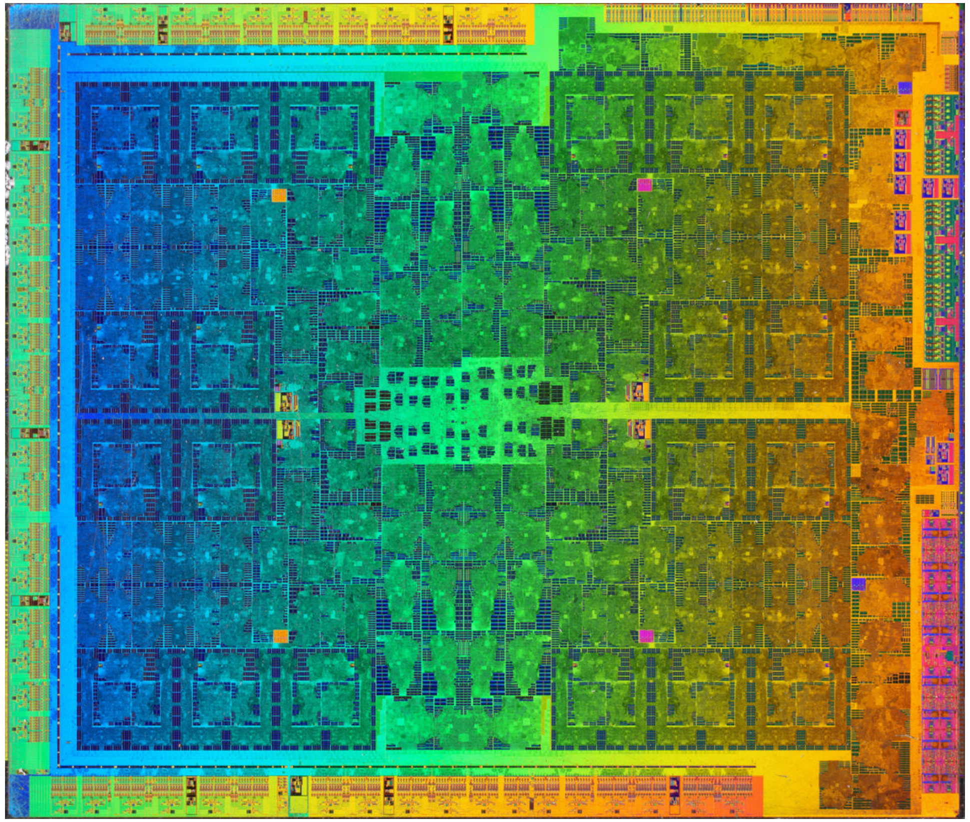


- 40% of the die is GPU
- 25% of the die is I/O
- 15% of the die is Cache

Only ~15% of the die is Compute

Focus on Latency

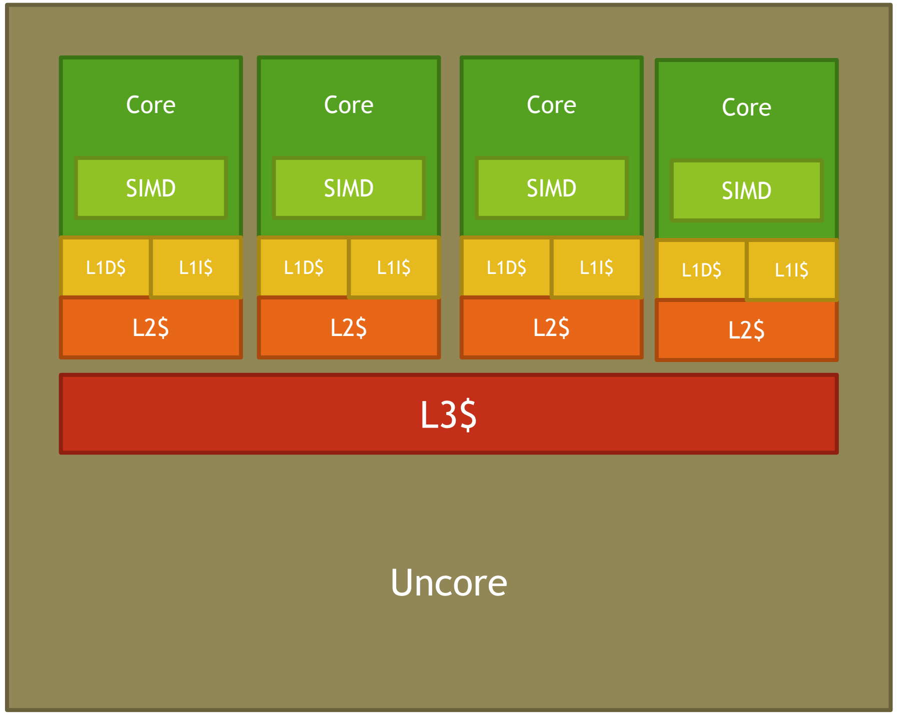
A Modern GPU (GTX 1070)



- Die Layout?
- **~70% is Compute**
- 10% Memory I/O
- 10% Registers
- 5% Cache

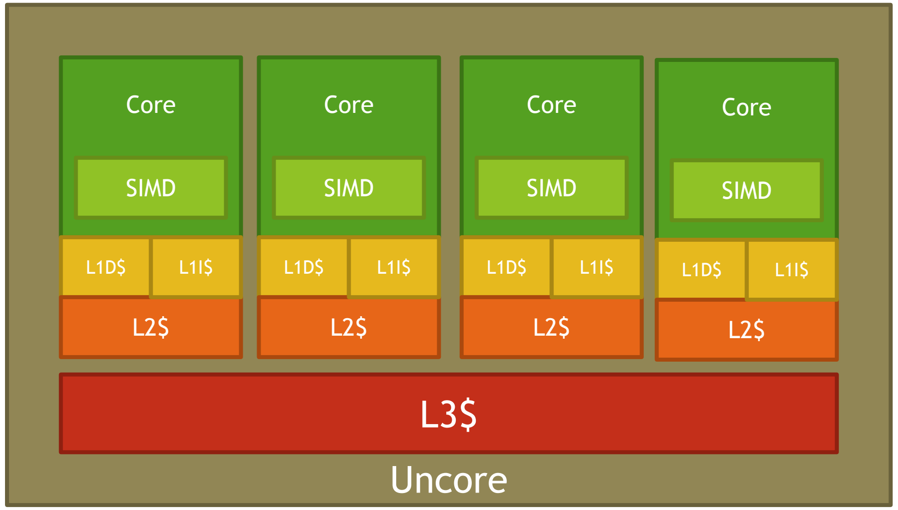
Focus on Throughput

Let's Convert a CPU to a GPU!



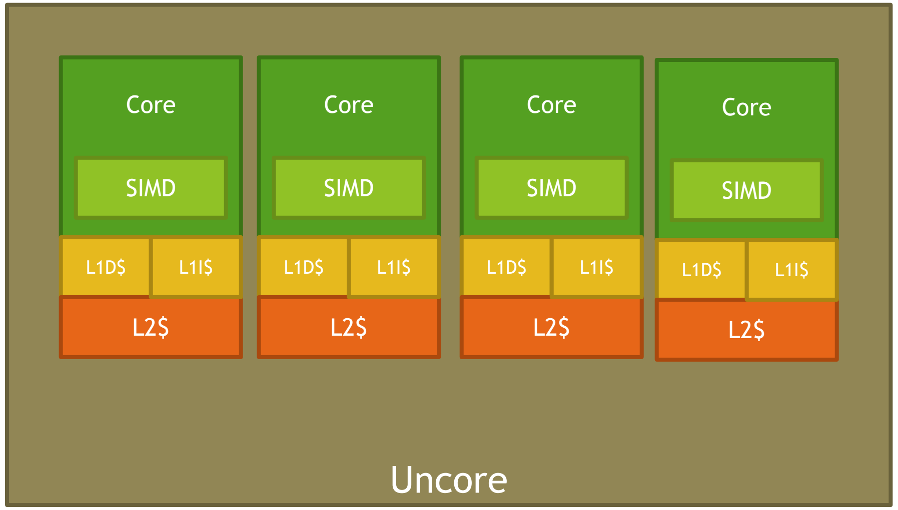
Step 1: Basic CPU

Let's Convert a CPU to a GPU!



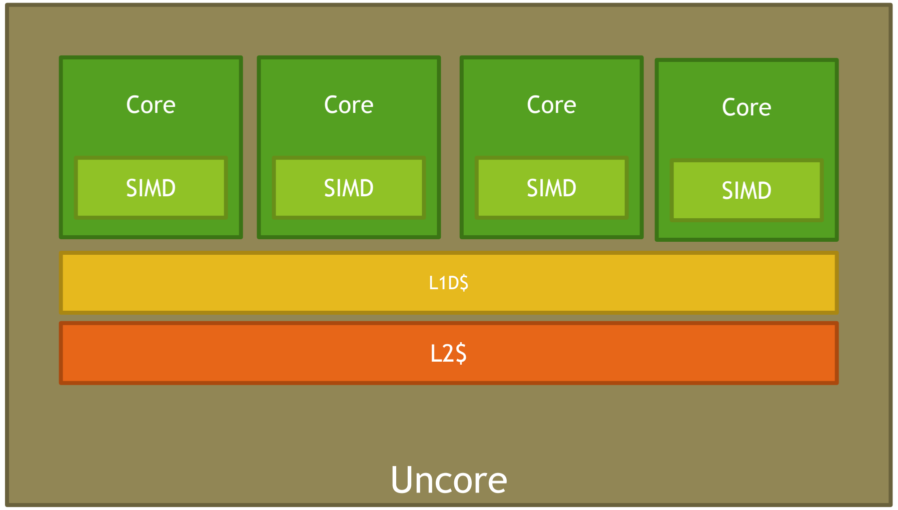
Step 2: Remove Unnecessary Uncore

Let’s Convert a CPU to a GPU!



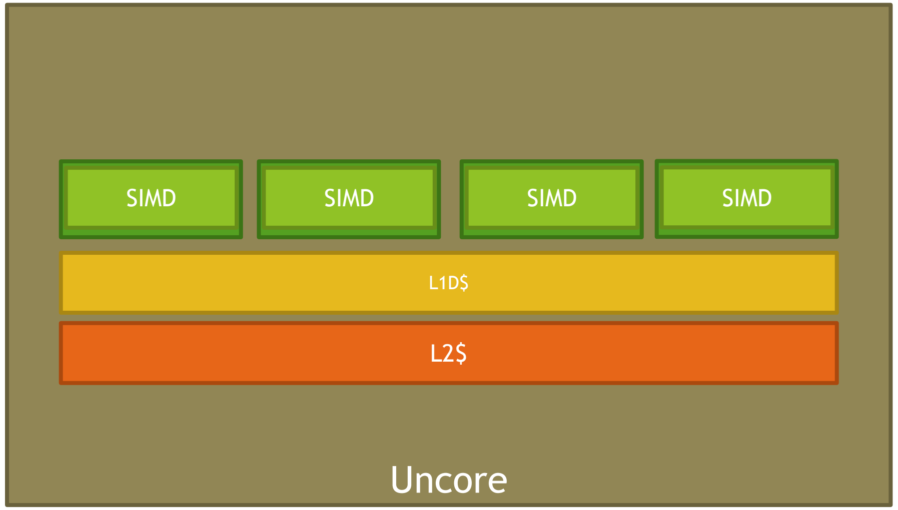
Step 3: Remove Outer (coherent) Cache

Let's Convert a CPU to a GPU!



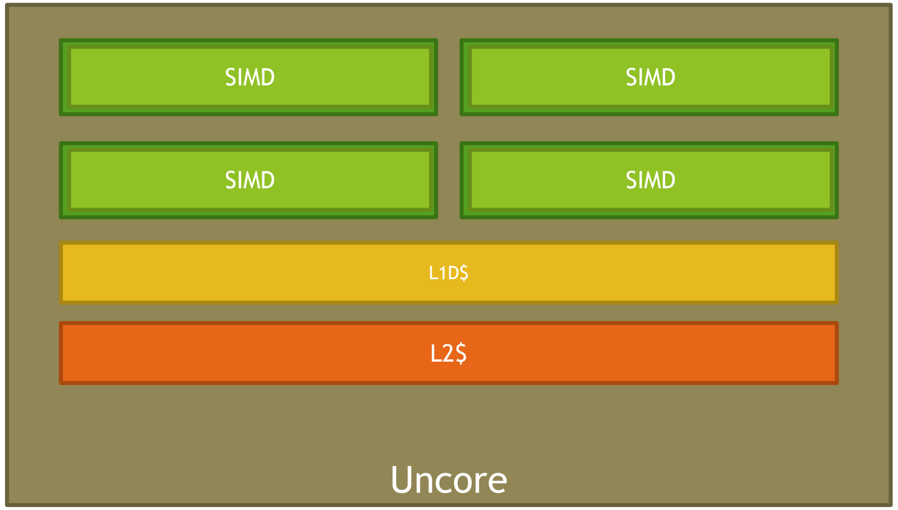
Step 4: Make L1 and L2 cache shared

Let’s Convert a CPU to a GPU!



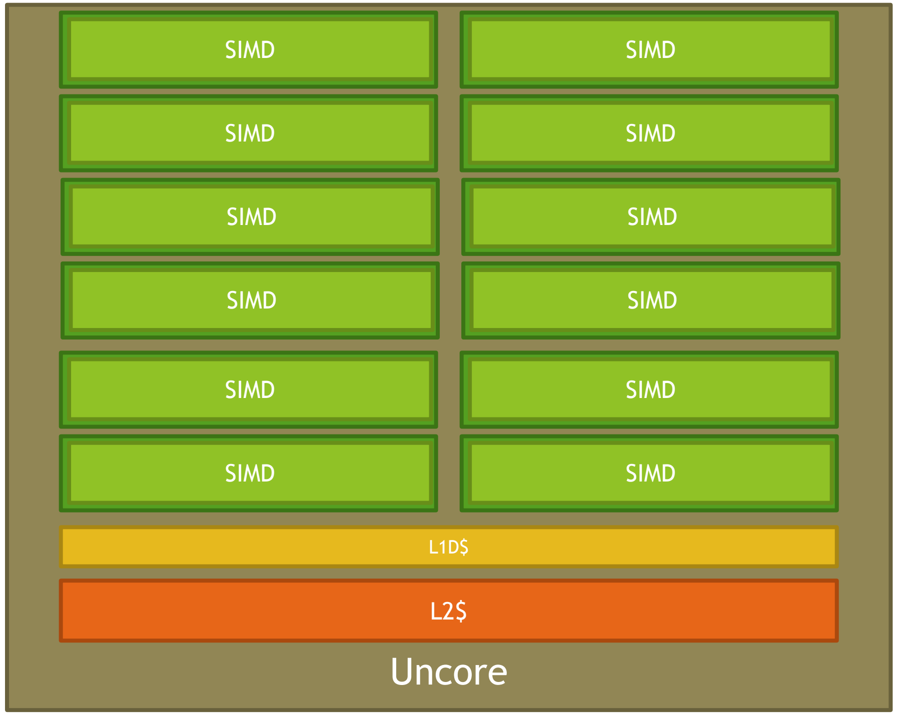
Step 5: Simplify Cores

Let’s Convert a CPU to a GPU!



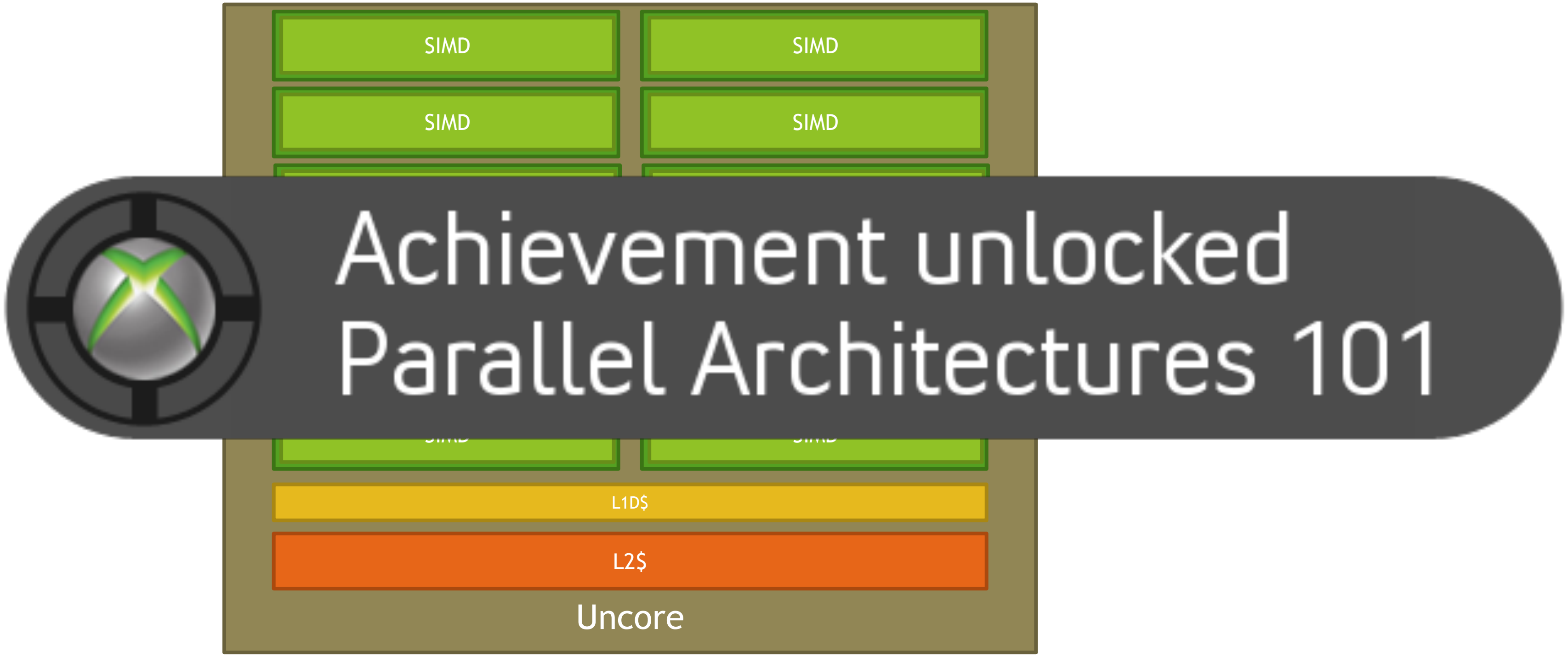
Step 5: Make SIMD Units Wider (4x)

Let's Convert a CPU to a GPU!



Step 6: Replicate Cores

Let’s Convert a CPU to a GPU!



Step 6: Replicate Cores

- Programming in Parallel *By Default*

Programming in Parallel *By Default*

- Challenges:
  - Identifying independence – what can/should be parallelized
  - Data management – data may not exist where we need it to be
  - Data hazards – modifying values potentially means overwriting
  - Programming model
    - ***How do we program for a parallel architecture***
    - How do we address the other challenges presented

Programming in Parallel *By Default*

- Case Study: Vector Addition

```
for (int i = 0; i < N; ++i)
  c[i] = a[i] + b[i];
```

  - Two source arrays (a, b)
  - One destination array

- Addressing our challenges:
  - Data independence?

Programming in Parallel *By Default*

- Directive-based Parallel Programming: SIMD

```
#pragma simd
for (int i = 0; i < N; ++i)
  c[i] = a[i] + b[i];
```

- The #pragma is a **hint** to the compiler to tell it that it can assume ”vector” independence. Iteration *k* does **not** depend on iteration *k-1*

- This is a good first step, but we are still only on the CPU
  - And still on one core!

Programming in Parallel *By Default*

- Directive-based Parallel Programming: OpenMP

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
  c[i] = a[i] + b[i];
```

- OpenMP is a programming model that allows a user to indicate what sections of code can be executed concurrently
- This is much better! We are now running on all cores of the CPU
  - But can we do more?

Programming in Parallel *By Default*

- Directive-based Parallel Programming: OpenMP with SIMD

```
#pragma omp parallel for simd
for (int i = 0; i < N; ++i)
  c[i] = a[i] + b[i];
```

  - We added the **simd** clause to the *directive*. This tells the compiler:
    - Parallelize across all **cores** with “omp parallel”
    - Parallelize across all **vector lanes** with “simd”
  - This is great! We are now saturating the CPU

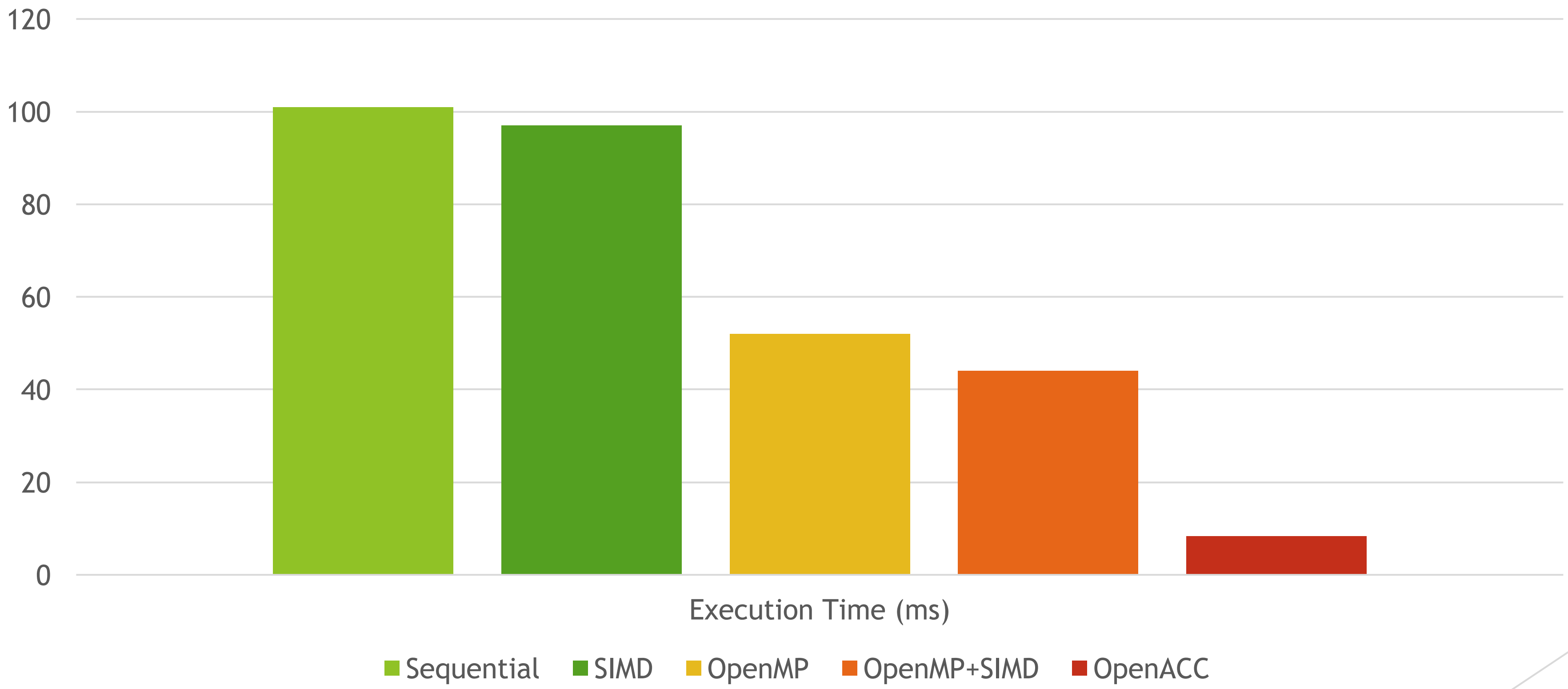
Programming in Parallel *By Default*

- Directive-based Parallel Programming: OpenACC

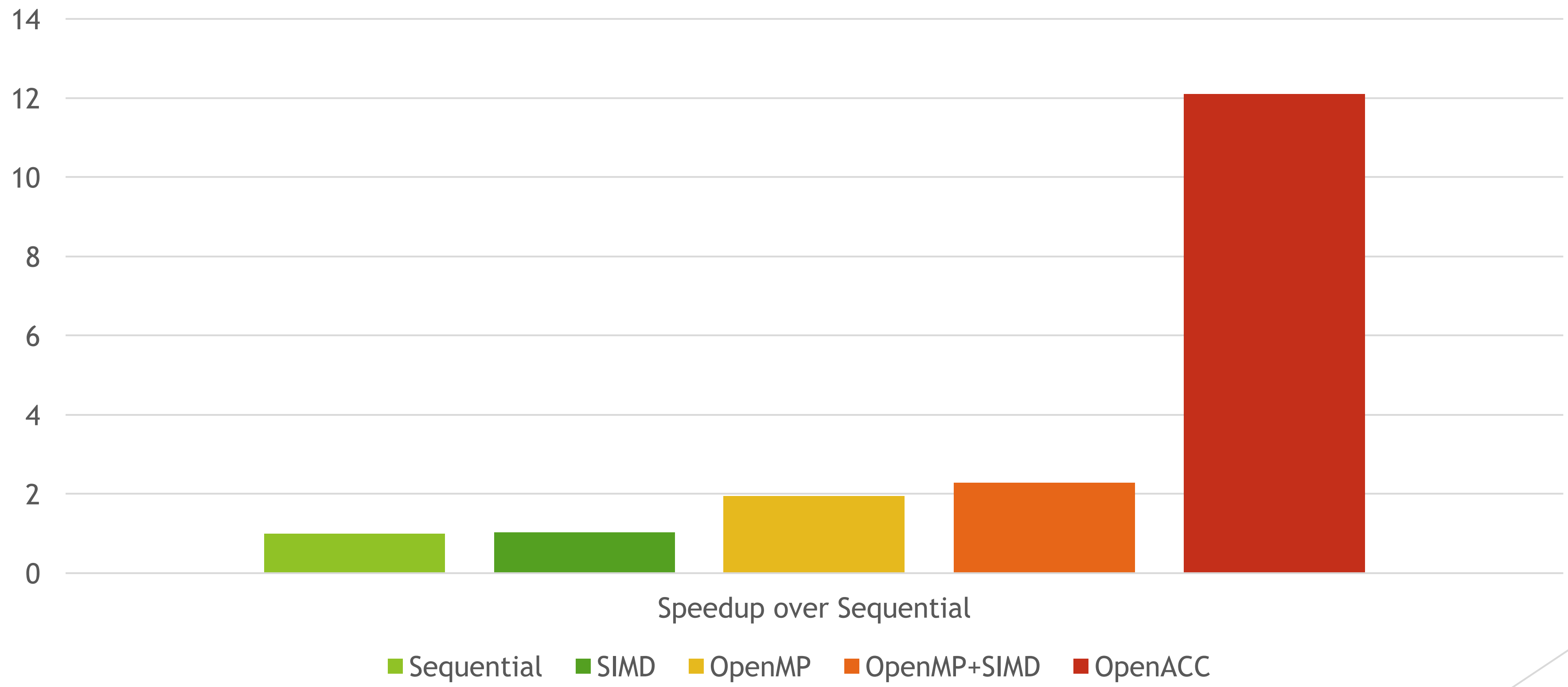
```
#pragma acc kernels
for (int i = 0; i < N; ++i)
  c[i] = a[i] + b[i];
```

- Woah, what happened?
  - OpenACC targets CPUs
  - Minimal source code change
  - Compiler analyzes your code
  - (Optionally) implicit data transfer
- But how well does it perform?

Vector Addition – Execution Time



Vector Addition – Speedup



Programming in Parallel *By Default*

- I didn't tell you everything though…
- With every **compiler**, there are **options** that you can give:
  - `g++ -std=c++11 –fopenmp -O3 –march=native vecadd.cpp –o vecadd`
    - `g++` **Compiler name**
    - `-std=c++11 –fopenmp` **Language flags**
    - `-O3 –march=native` **Optimization flags**
    - `vecadd.cpp` **Source file**
    - `-o vecadd` **Output file**

Programming in Parallel *By Default*

- Let's have a look at what options I had to give the OpenACC compiler
  - PGI Community Edition 16.10
  - `pgc++ -std=c++11 -acc -ta=tesla:managed,cc50 -O3 vecadd.cpp -o vecadd`
    - `pgc++` **Compiler name**
    - `-std=c++11 -acc` **Language flags (-acc enables OpenACC)**
    - `-ta=tesla:managed,cc50 -O3` **Automatic memory transfer, target GPU, optimize**
    - `vecadd.cpp` **Source file**
    - `-o vecadd` **Output file**

Memory Management

- Leveraging the Power of GPUs
- Data that you normally create is:
  - Available for use on the CPU you are running on
  - Not available anywhere else
- What does this mean for the programmer?
  - They need to get the data onto the GPU
  - … And back!

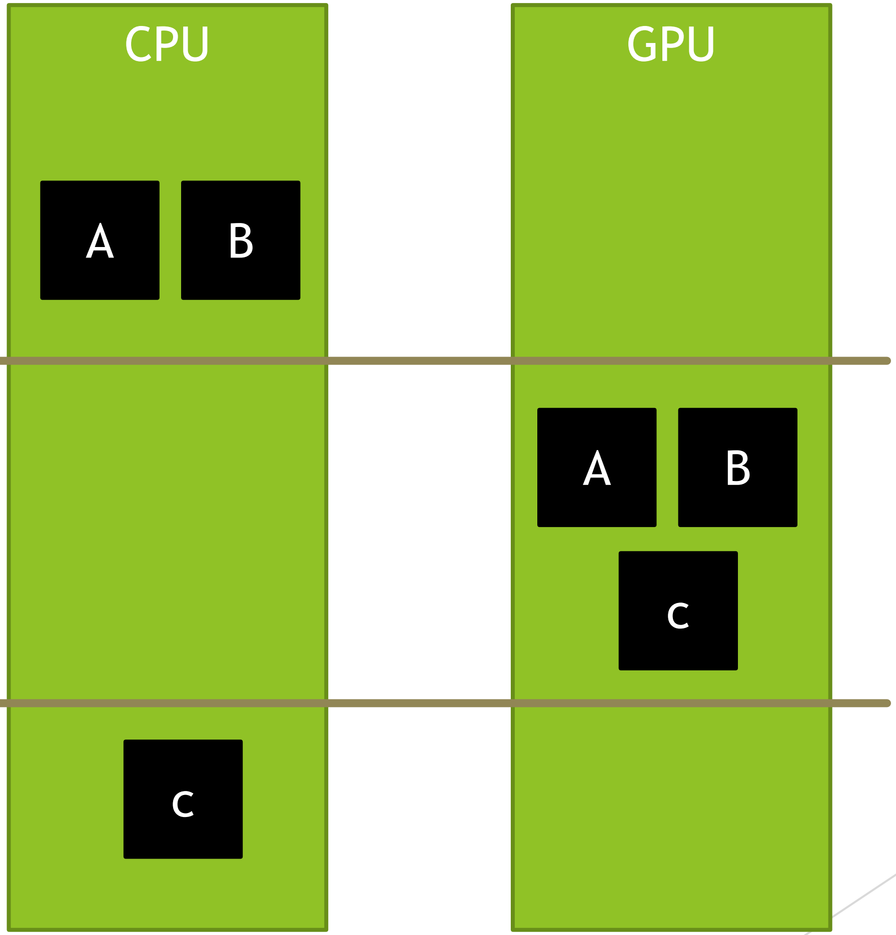
Automatic Memory Management

Source Code

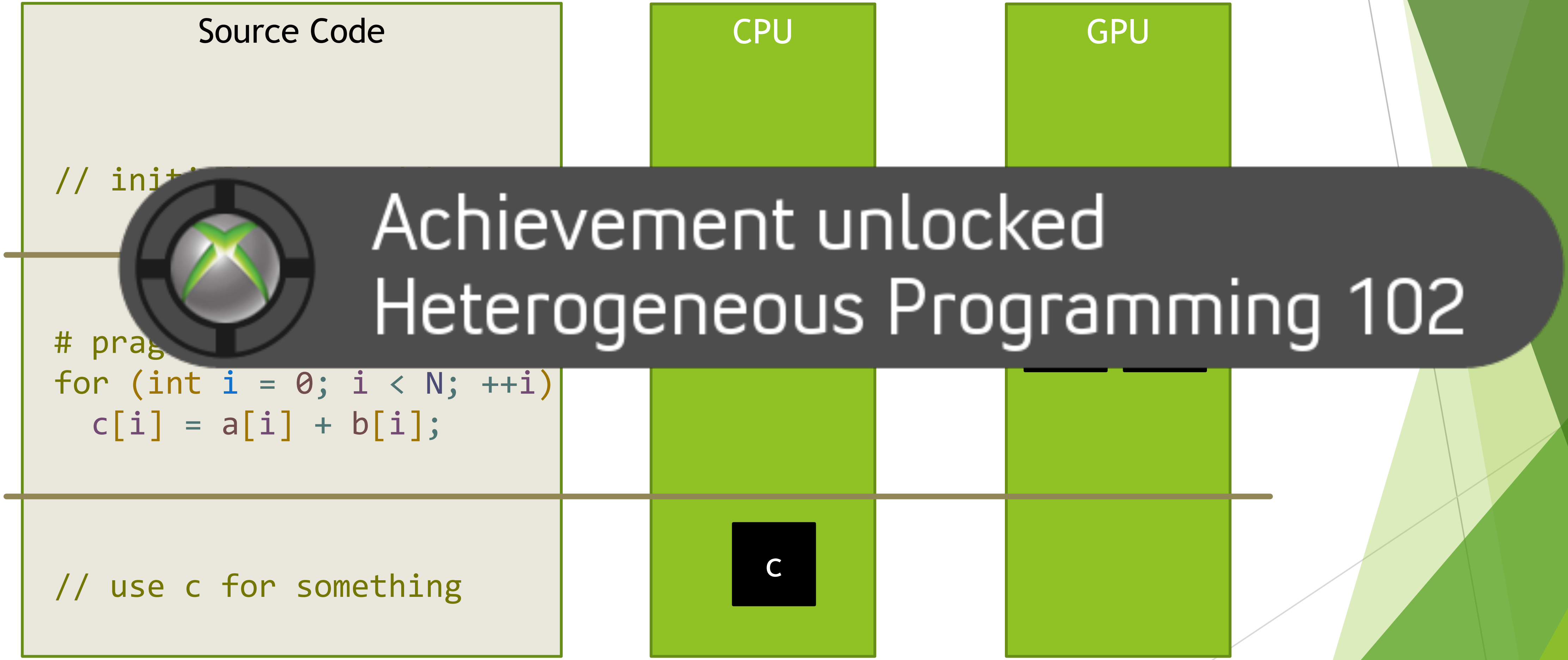
```
// initialize a and b

# pragma acc kernels
for (int i = 0; i < N; ++i)
  c[i] = a[i] + b[i];

// use c for something
```

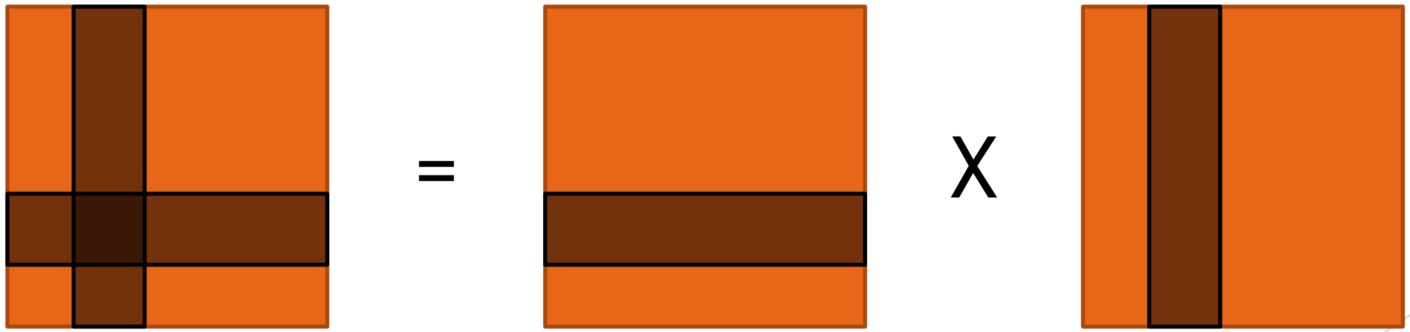


Automatic Memory Management



Case Study 2: Matrix Multiply

- Commonly used in:
  - Computer Graphics
  - Physics Modeling/Simulation
  - Linear Algebra Routines
- Computationally Expensive: $O(N^3)$
- Storage Costs Relatively High: $O(N^2)$



Case Study 2: Matrix Multiply

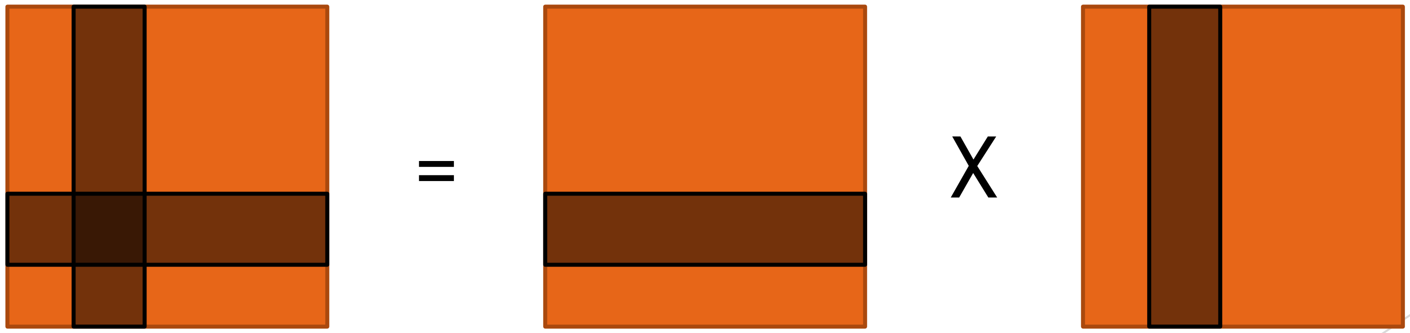
Live Demo – Interactive Terminal

GitHub Repository

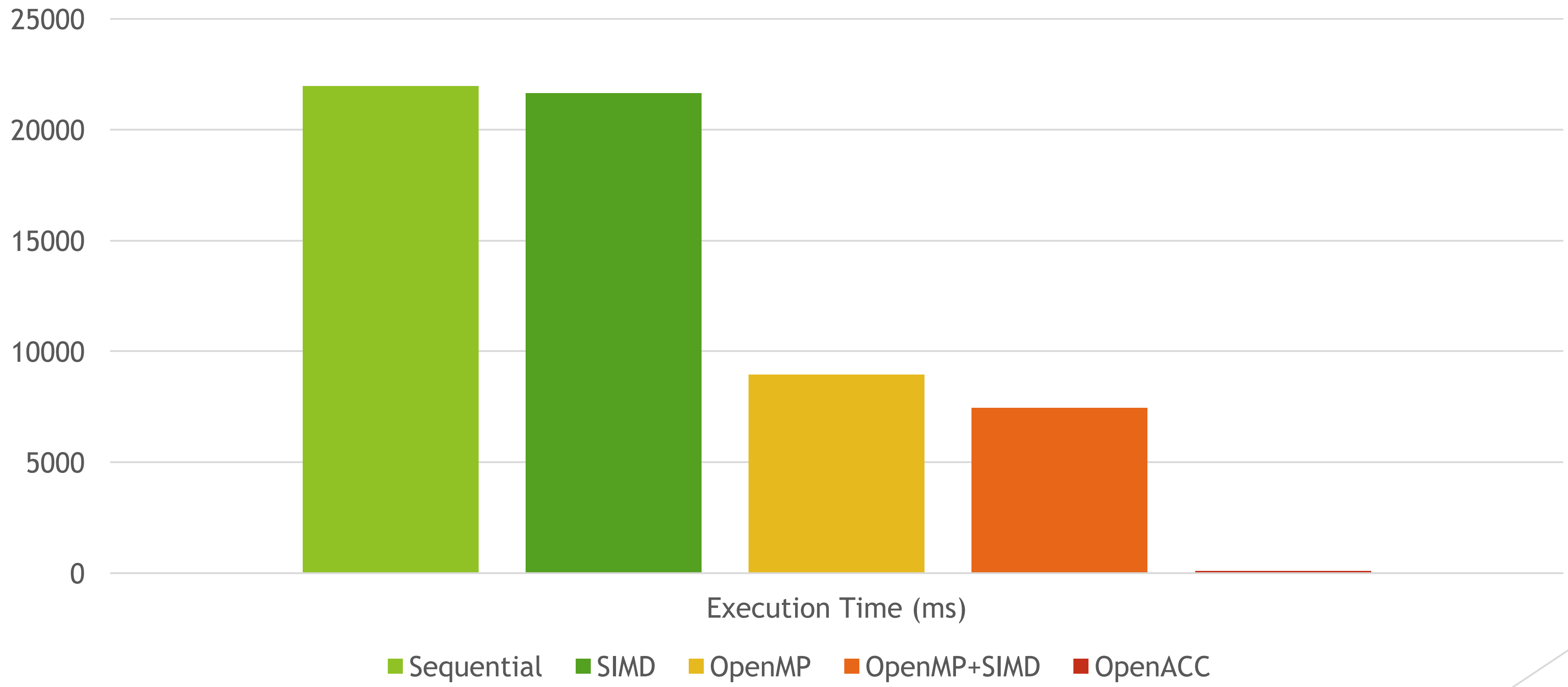
https://github.com/willkill07/gpu-programming-intro

Asciinema Recording (check back later)

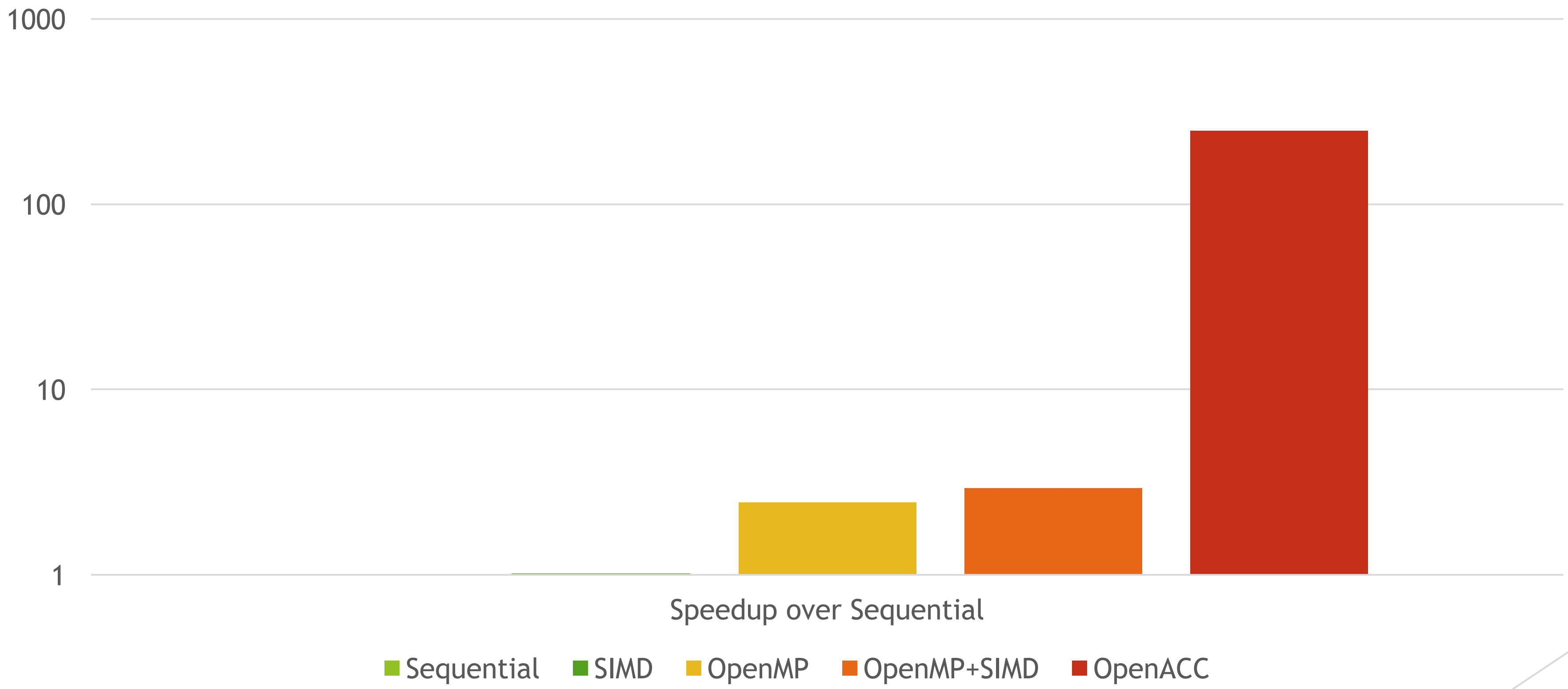
https://asciinema.org/~willkill07



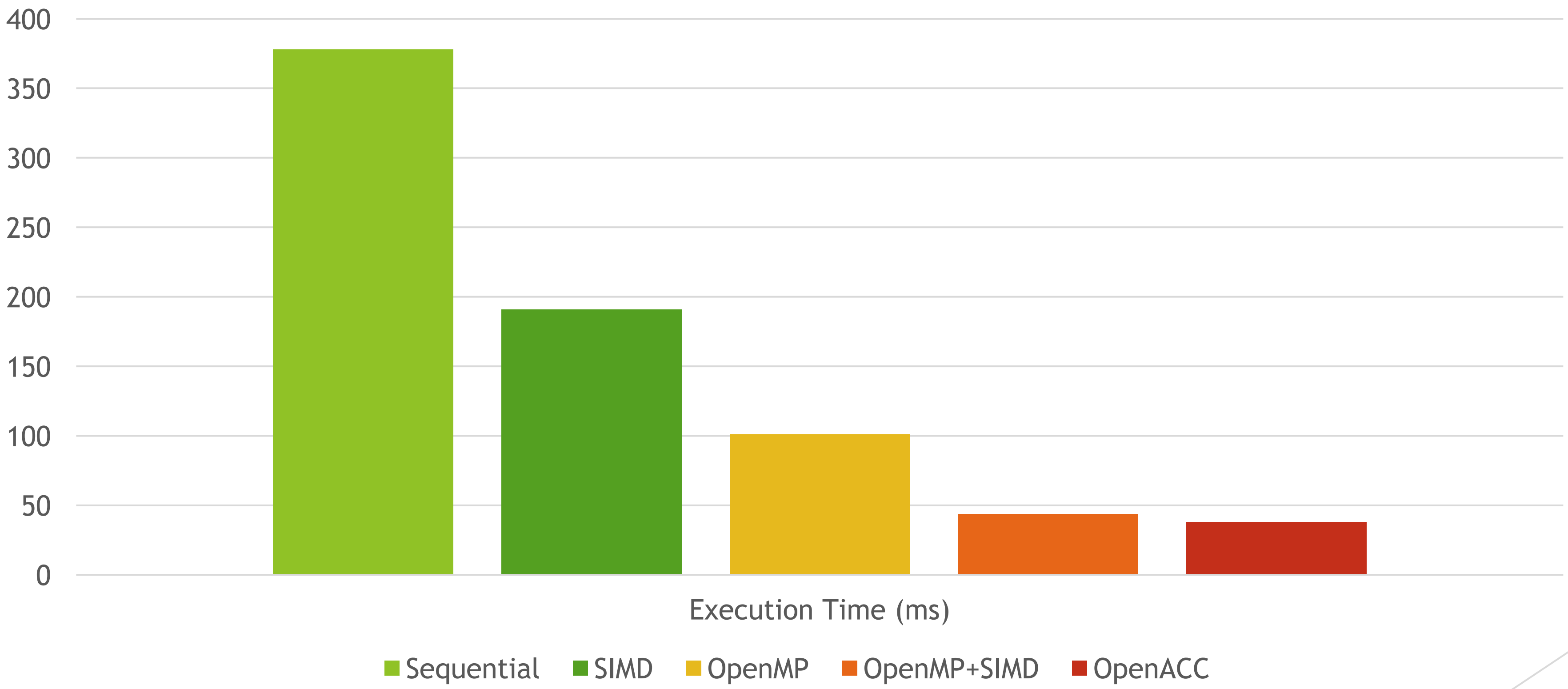
Matrix Multiplication – Execution Time



Matrix Multiplication – Speedup



2D Stencil – Execution Time



2D Stencil – Speedup