# GPU programming 101

*Cedric Nugteren*

Amsterdam C++ meetup
2016 - 08 - 25

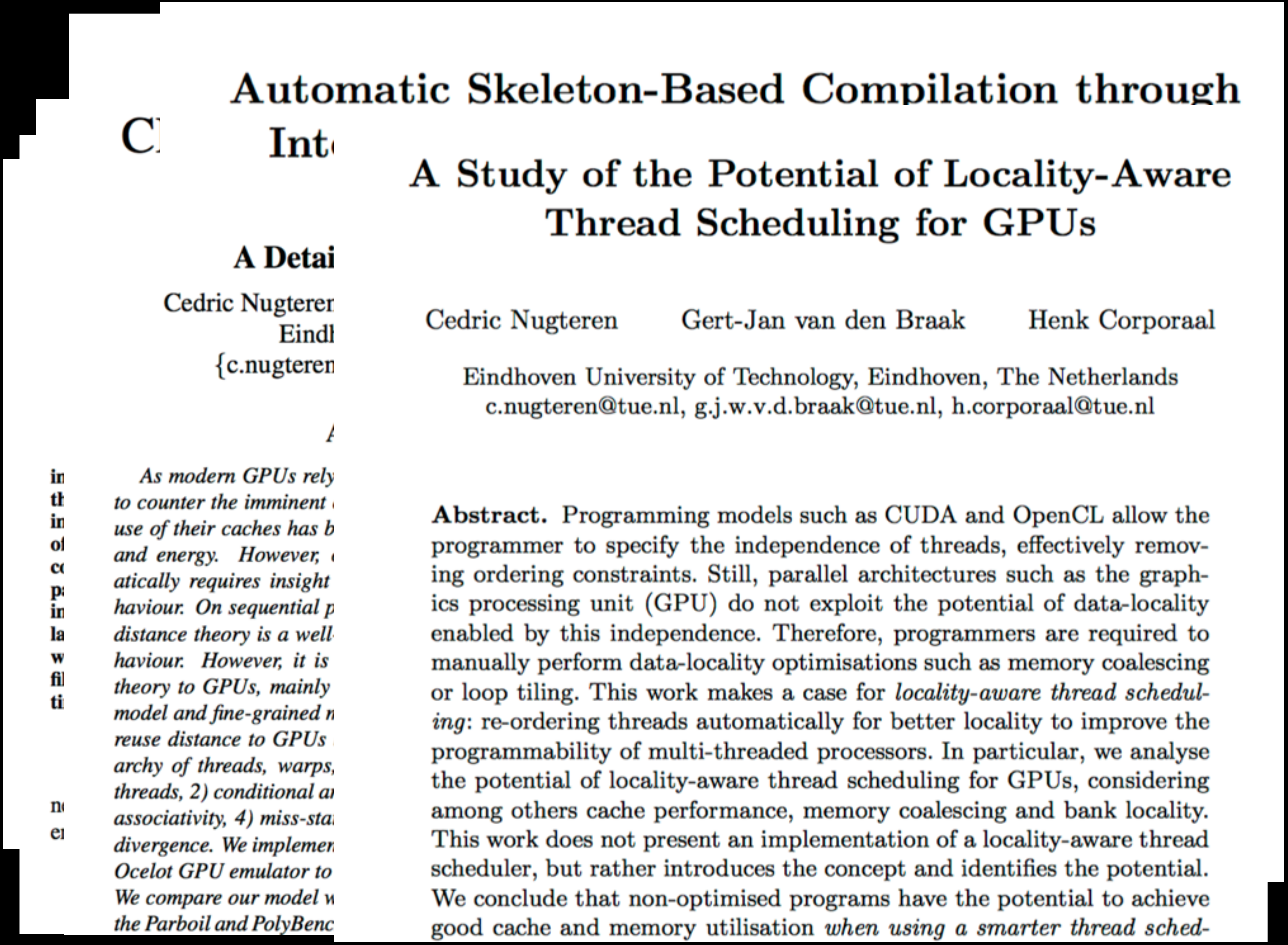
Hercules 3D GRAPHICS
128MB DDR AGP
3D PROPHET 9500 PRO
WinFast GeForce 256
Unparalleled 256-bit Graphics Processing Unit
Fulfill QuadEngine™ Technology
Integrated 350Mhz RAMDAC
Digital Flat Panel & NTSC/PAL TV-Output (Option)
Fully support Microsoft DirectX 7.0 & OpenGL 1.2 ICD
Optimized for Intel Pentium III SSE and AMD 3DNow!
Leadtek
ASUS
AGP-V6600 Brings Graphics into New Era
GeForce256 32 MB
Powered with GeForce256™ GPU
4x4 engine with Transform and Lighting integrated
Smart Doctor™ technologies provide the best safety for your valuable system
AGP 4X with Fast Writes
ASUS DVD software player
Optimized for DirectX®7 & OpenGL®
Hercules
3D PROPHET DDR
CREATIVE
3D BLASTER
GeForce256 ANNIHILATOR PRO
32MB DDR Memory - The Power to Play Games the Way You Want!
Deluxe
GeForce256 32 MB
HARDWARE INFO

TITAN
GEFORCE GTX TITAN X
INSPIRED BY GAMERS. BUILT BY NVIDIA.

# Agenda

1. Intro GPU architecture
2. Intro GPU programming model
3. CUDA/OpenCL by example: matrix-multiplication
4. C++11 ❤️ GPU → SyCL

Why believe me?



Automatic Skeleton-Based Compilation through

Int

Cl

A Detai

Cedric Nugterer

Eindl

{c.nugteren

in
th
in
of
c
p
in
la
w
fi
ti

n
e

As modern GPUs rely
to counter the imminent
use of their caches has b
and energy. However,
atically requires insight
haviour. On sequential p
distance theory is a well
haviour. However, it is
theory to GPUs, mainly
model and fine-grained n
reuse distance to GPUs
archy of threads, warps,
threads, 2) conditional a
associativity, 4) miss-sta
divergence. We implemen
Ocelot GPU emulator to
We compare our model w
the Parboil and PolyBenc

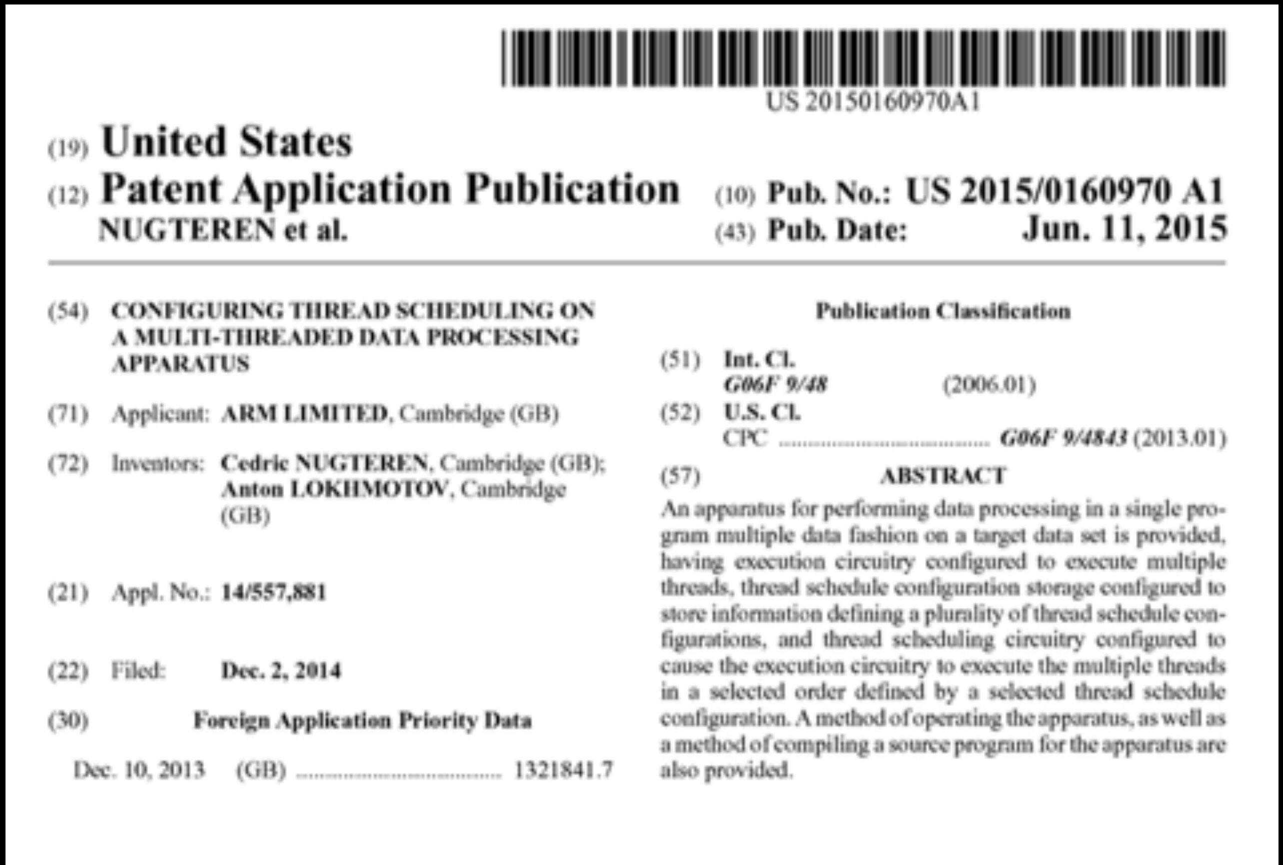
# A Study of the Potential of Locality-Aware Thread Scheduling for GPUs

Cedric Nugteren Gert-Jan van den Braak Henk Corporaal

Eindhoven University of Technology, Eindhoven, The Netherlands
c.nugteren@tue.nl, g.j.w.v.d.braak@tue.nl, h.corporaal@tue.nl

**Abstract.** Programming models such as CUDA and OpenCL allow the programmer to specify the independence of threads, effectively removing ordering constraints. Still, parallel architectures such as the graphics processing unit (GPU) do not exploit the potential of data-locality enabled by this independence. Therefore, programmers are required to manually perform data-locality optimisations such as memory coalescing or loop tiling. This work makes a case for *locality-aware thread scheduling*: re-ordering threads automatically for better locality to improve the programmability of multi-threaded processors. In particular, we analyse the potential of locality-aware thread scheduling for GPUs, considering among others cache performance, memory coalescing and bank locality. This work does not present an implementation of a locality-aware thread scheduler, but rather introduces the concept and identifies the potential. We conclude that non-optimised programs have the potential to achieve good cache and memory utilisation *when using a smarter thread sched-*

TU/e Technische Universiteit
**Eindhoven**
University of Technology



US 20150160970A1

(19) United States

(12) Patent Application Publication

NUGTEREN et al.

(10) Pub. No.: US 2015/0160970 A1

(43) Pub. Date: Jun. 11, 2015

(54) CONFIGURING THREAD SCHEDULING ON A MULTI-THREADED DATA PROCESSING APPARATUS

(71) Applicant: ARM LIMITED, Cambridge (GB)

(72) Inventors: Cedric NUGTEREN, Cambridge (GB); Anton LOKHMOTOV, Cambridge (GB)

(21) Appl. No.: 14/557,881

(22) Filed: Dec. 2, 2014

(30) Foreign Application Priority Data

Dec. 10, 2013 (GB) .................................. 1321841.7

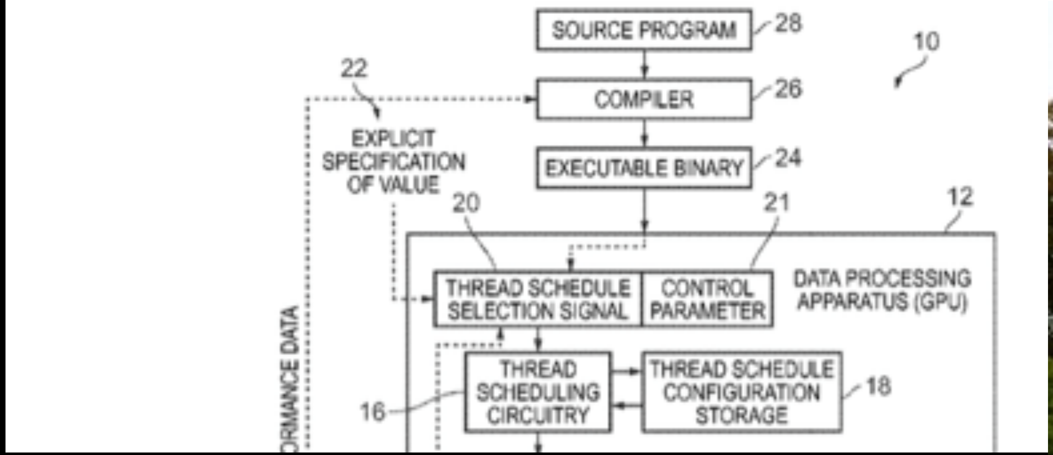
Publication Classification

(51) Int. Cl.
*G06F 9/48* (2006.01)

(52) U.S. Cl.
CPC .................................. *G06F 9/4843* (2013.01)

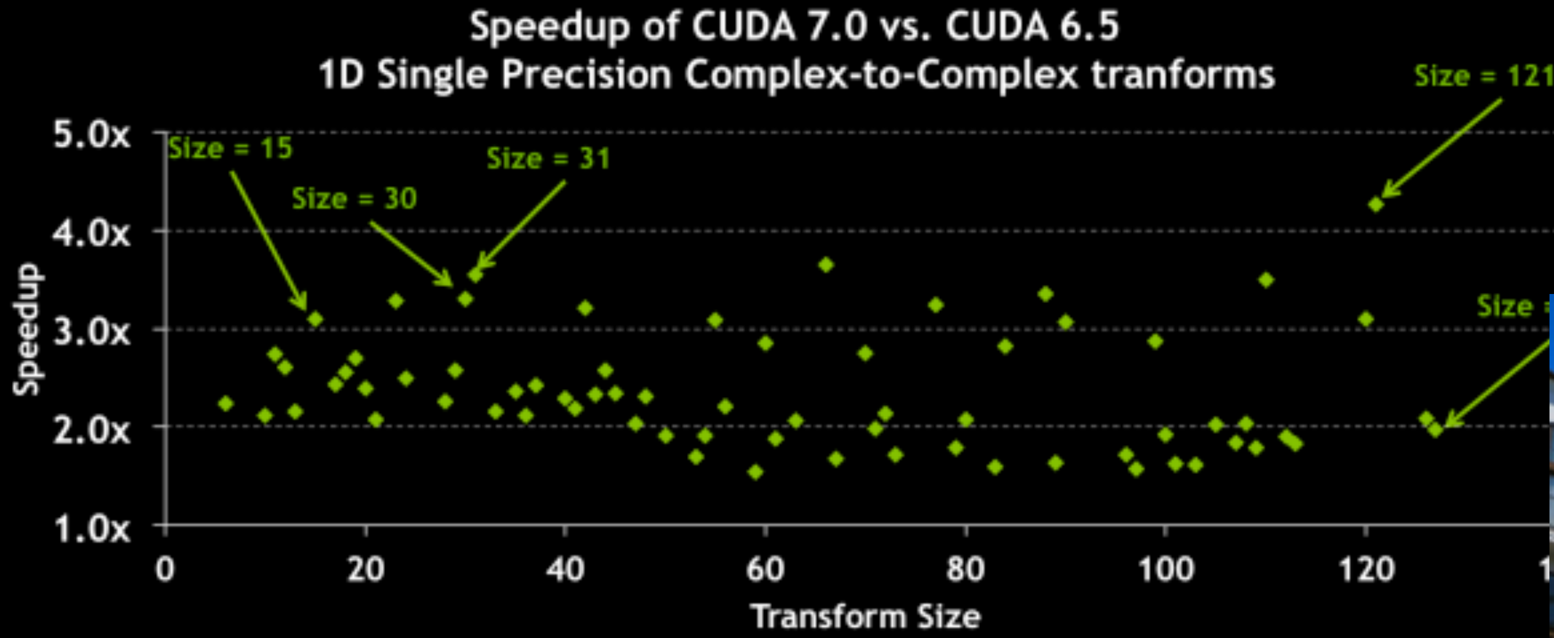
(57) ABSTRACT

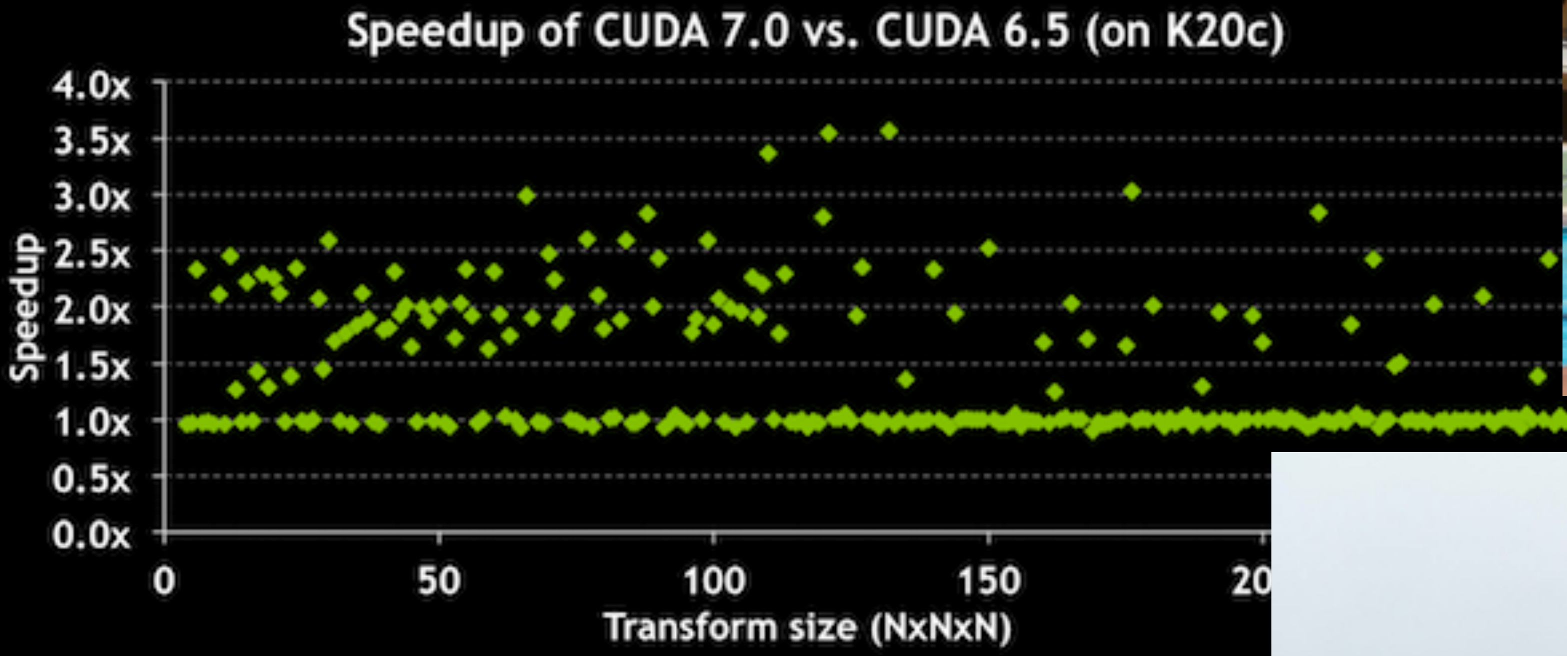
An apparatus for performing data processing in a single program multiple data fashion on a target data set is provided, having execution circuitry configured to execute multiple threads, thread schedule configuration storage configured to store information defining a plurality of thread schedule configurations, and thread scheduling circuitry configured to cause the execution circuitry to execute the multiple threads in a selected order defined by a selected thread schedule configuration. A method of operating the apparatus, as well as a method of compiling a source program for the apparatus are also provided.


SOURCE PROGRAM
28
10
COMPILER
26
22
EXPLICIT SPECIFICATION OF VALUE
EXECUTABLE BINARY
24
20
21
12
THREAD SCHEDULE SELECTION SIGNAL
CONTROL PARAMETER
DATA PROCESSING APPARATUS (GPU)
ORMANCE DATA
THREAD SCHEDULING CIRCUITRY
THREAD SCHEDULE CONFIGURATION STORAGE
16
18

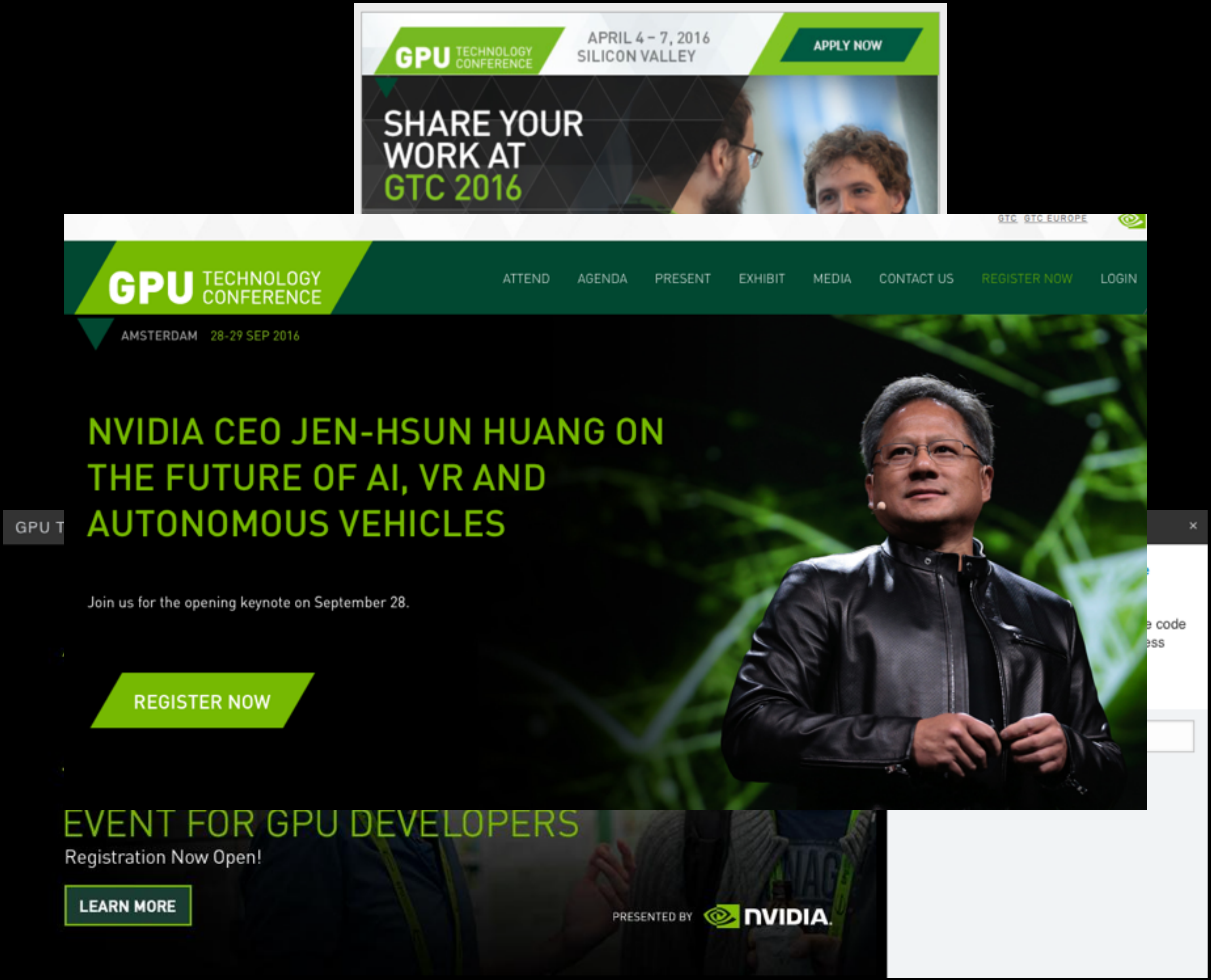
OpenCL

NVIDIA
CUDA


Speedup of CUDA 7.0 vs. CUDA 6.5
1D Single Precision Complex-to-Complex tranforms
Size = 121
5.0x
Size = 15
Size = 30
Size = 31
4.0x
Size =
Speedup
3.0x
2.0x
1.0x
0
20
40
60
80
100
120
1
Transform Size


Speedup of CUDA 7.0 vs. CUDA 6.5 (on K20c)
4.0x
3.5x
3.0x
2.5x
2.0x
1.5x
1.0x
0.5x
0.0x
Speedup
0
50
100
150
20
Transform size (NxNxN)

NVIDIA
NVIDIA
NVIDIA
EVGA
TEGRA NOTE 7
NVIDIA
NVIDIA
NVIDIA
GPU TECHNOLOGY CONFERENCE



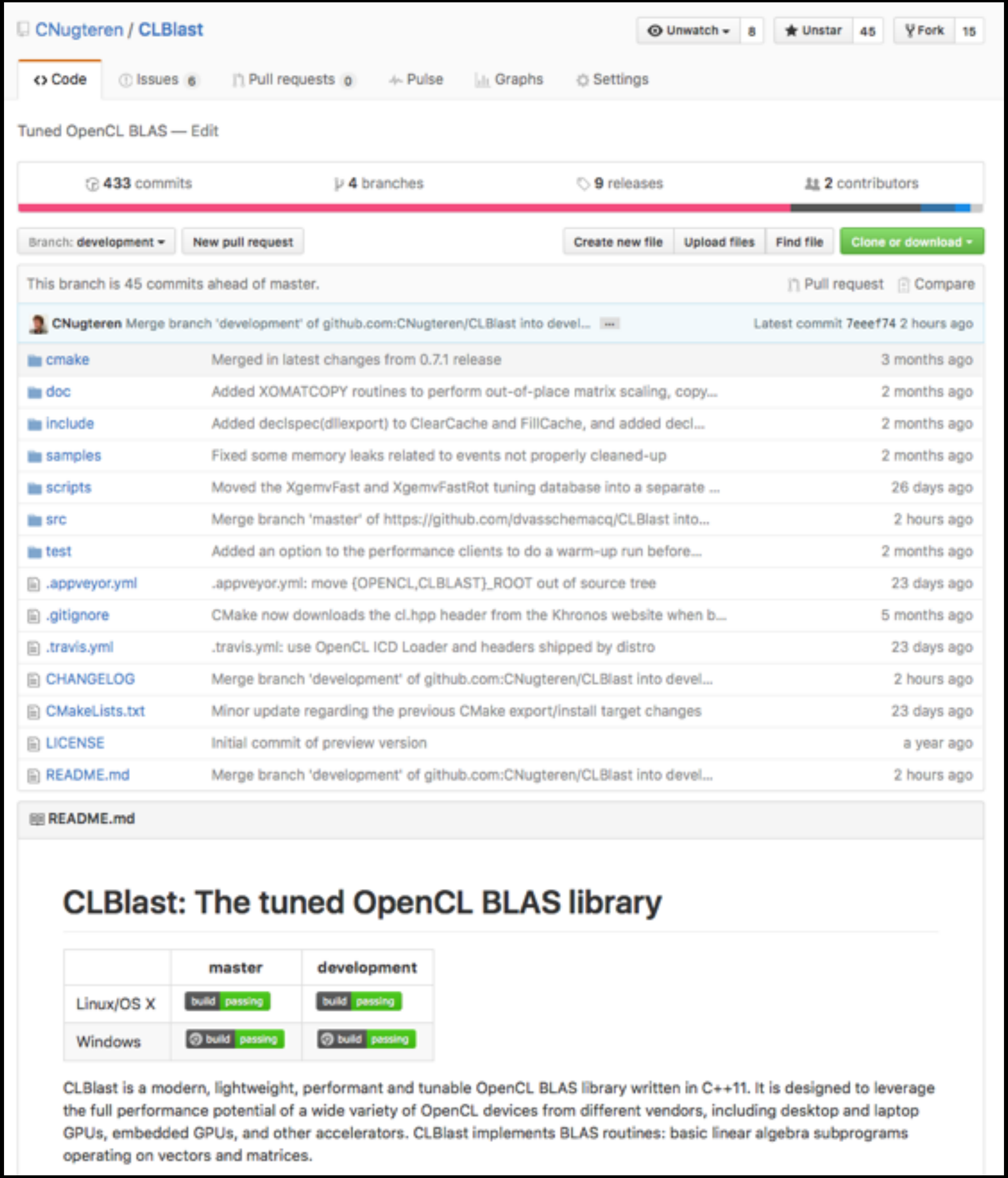
SHARE YOUR WORK AT GTC 2016

NVIDIA CEO JEN-HSUN HUANG ON THE FUTURE OF AI, VR AND AUTONOMOUS VEHICLES

Join us for the opening keynote on September 28.

EVENT FOR GPU DEVELOPERS

Registration Now Open!



CNugteren / CLBlast

Tuned OpenCL BLAS — Edit

433 commits   4 branches   9 releases   2 contributors

This branch is 45 commits ahead of master.

CNugteren Merge branch 'development' of github.com:CNugteren/CLBlast into devel... Latest commit 7eeef74 2 hours ago

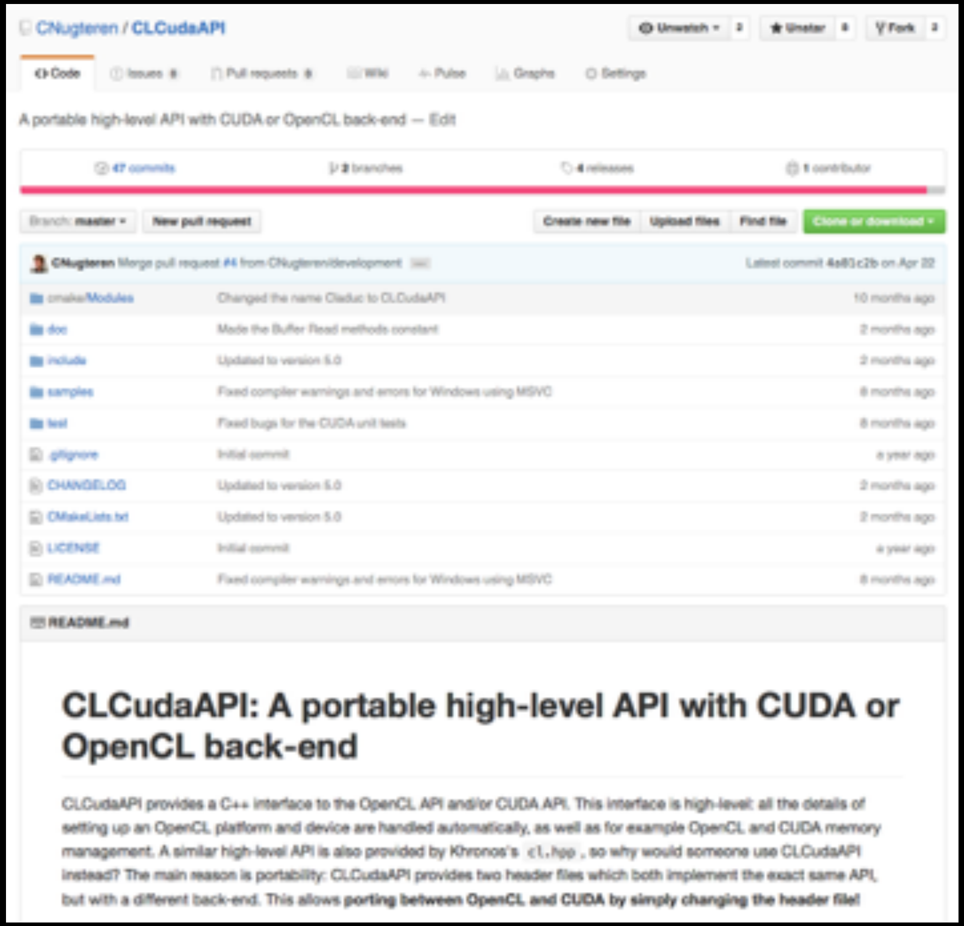
| | | |
|---|---|---|
| cmake | Merged in latest changes from 0.7.1 release | 3 months ago |
| doc | Added XOMATCOPY routines to perform out-of-place matrix scaling, copy... | 2 months ago |
| include | Added declspec(dllexport) to ClearCache and FillCache, and added decl... | 2 months ago |
| samples | Fixed some memory leaks related to events not properly cleaned-up | 2 months ago |
| scripts | Moved the XgemvFast and XgemvFastRot tuning database into a separate ... | 26 days ago |
| src | Merge branch 'master' of https://github.com/dvasschemacq/CLBlast into... | 2 hours ago |
| test | Added an option to the performance clients to do a warm-up run before... | 2 months ago |
| .appveyor.yml | .appveyor.yml: move {OPENCL,CLBLAST}_ROOT out of source tree | 23 days ago |
| .gitignore | CMake now downloads the cl.hpp header from the Khronos website when b... | 5 months ago |
| .travis.yml | .travis.yml: use OpenCL ICD Loader and headers shipped by distro | 23 days ago |
| CHANGELOG | Merge branch 'development' of github.com:CNugteren/CLBlast into devel... | 2 hours ago |
| CMakeLists.txt | Minor update regarding the previous CMake export/install target changes | 23 days ago |
| LICENSE | Initial commit of preview version | a year ago |
| README.md | Merge branch 'development' of github.com:CNugteren/CLBlast into devel... | 2 hours ago |

README.md

# CLBlast: The tuned OpenCL BLAS library

| | master | development |
|---|---|---|
| Linux/OS X | build passing | build passing |
| Windows | build passing | build passing |

CLBlast is a modern, lightweight, performant and tunable OpenCL BLAS library written in C++11. It is designed to leverage the full performance potential of a wide variety of OpenCL devices from different vendors, including desktop and laptop GPUs, embedded GPUs, and other accelerators. CLBlast implements BLAS routines: basic linear algebra subprograms operating on vectors and matrices.



CNugteren / CLCudaAPI

A portable high-level API with CUDA or OpenCL back-end — Edit

47 commits   2 branches   4 releases   1 contributor

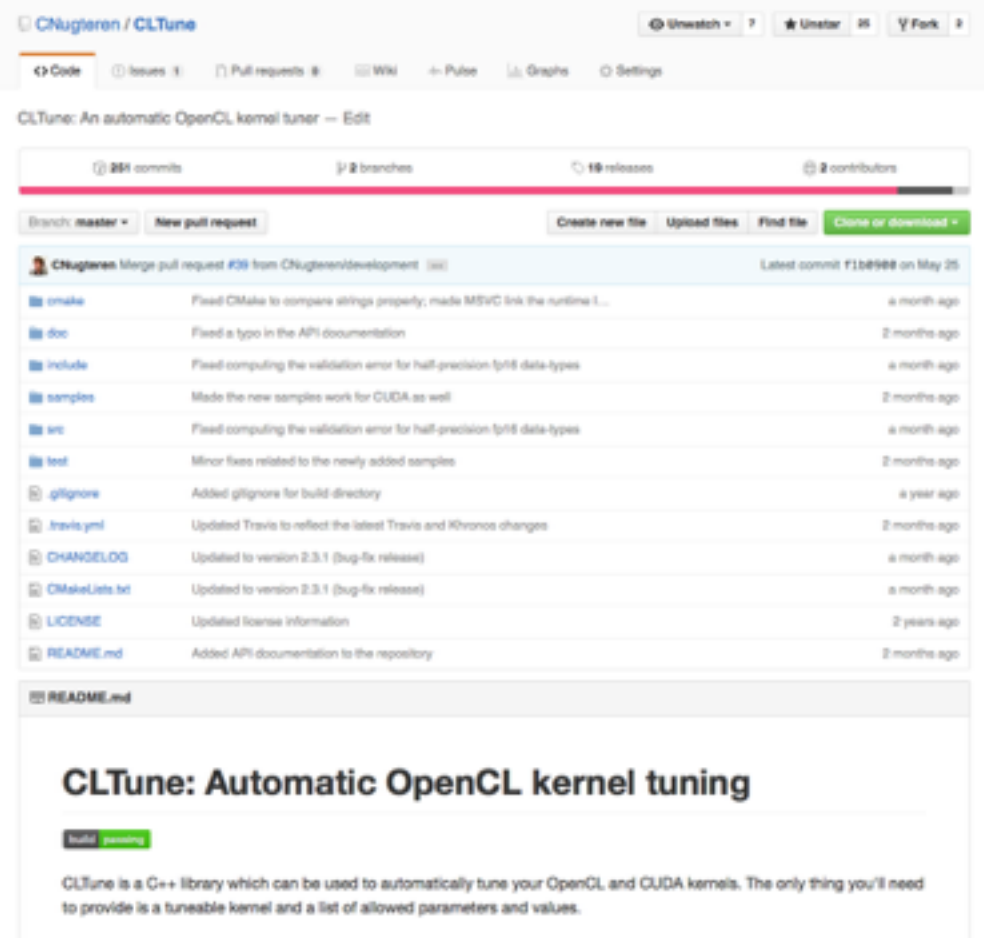
CNugteren Merge pull request #4 from CNugteren/development Latest commit 4a81c2b on Apr 22

| | | |
|---|---|---|
| cmake/Modules | Changed the name Claduc to CLCudaAPI | 10 months ago |
| doc | Made the Buffer Read methods constant | 2 months ago |
| include | Updated to version 5.0 | 2 months ago |
| samples | Fixed compiler warnings and errors for Windows using MSVC | 8 months ago |
| test | Fixed bugs for the CUDA unit tests | 8 months ago |
| .gitignore | Initial commit | a year ago |
| CHANGELOG | Updated to version 5.0 | 2 months ago |
| CMakeLists.txt | Updated to version 5.0 | 2 months ago |
| LICENSE | Initial commit | a year ago |
| README.md | Fixed compiler warnings and errors for Windows using MSVC | 8 months ago |

README.md

# CLCudaAPI: A portable high-level API with CUDA or OpenCL back-end

CLCudaAPI provides a C++ interface to the OpenCL API and/or CUDA API. This interface is high-level: all the details of setting up an OpenCL platform and device are handled automatically, as well as for example OpenCL and CUDA memory management. A similar high-level API is also provided by Khronos's `cl.hpp`, so why would someone use CLCudaAPI instead? The main reason is portability: CLCudaAPI provides two header files which both implement the exact same API, but with a different back-end. This allows **porting between OpenCL and CUDA by simply changing the header file!**



CNugteren / CLTune

CLTune: An automatic OpenCL kernel tuner — Edit

251 commits   2 branches   19 releases   2 contributors

CNugteren Merge pull request #39 from CNugteren/development Latest commit f1b0908 on May 25

| | | |
|---|---|---|
| cmake | Fixed CMake to compare strings properly; made MSVC link the runtime l... | a month ago |
| doc | Fixed a typo in the API documentation | 2 months ago |
| include | Fixed computing the validation error for half-precision fp16 data-types | a month ago |
| samples | Made the new samples work for CUDA as well | 2 months ago |
| src | Fixed computing the validation error for half-precision fp16 data-types | a month ago |
| test | Minor fixes related to the newly added samples | 2 months ago |
| .gitignore | Added gitignore for build directory | a year ago |
| .travis.yml | Updated Travis to reflect the latest Travis and Khronos changes | 2 months ago |
| CHANGELOG | Updated to version 2.3.1 (bug-fix release) | a month ago |
| CMakeLists.txt | Updated to version 2.3.1 (bug-fix release) | a month ago |
| LICENSE | Updated license information | 2 years ago |
| README.md | Added API documentation to the repository | 2 months ago |

README.md

# CLTune: Automatic OpenCL kernel tuning

build passing

CLTune is a C++ library which can be used to automatically tune your OpenCL and CUDA kernels. The only thing you'll need to provide is a tuneable kernel and a list of allowed parameters and values.

## Cartesius supercomputer

SURF SARA

## Embedded systems (mobile)

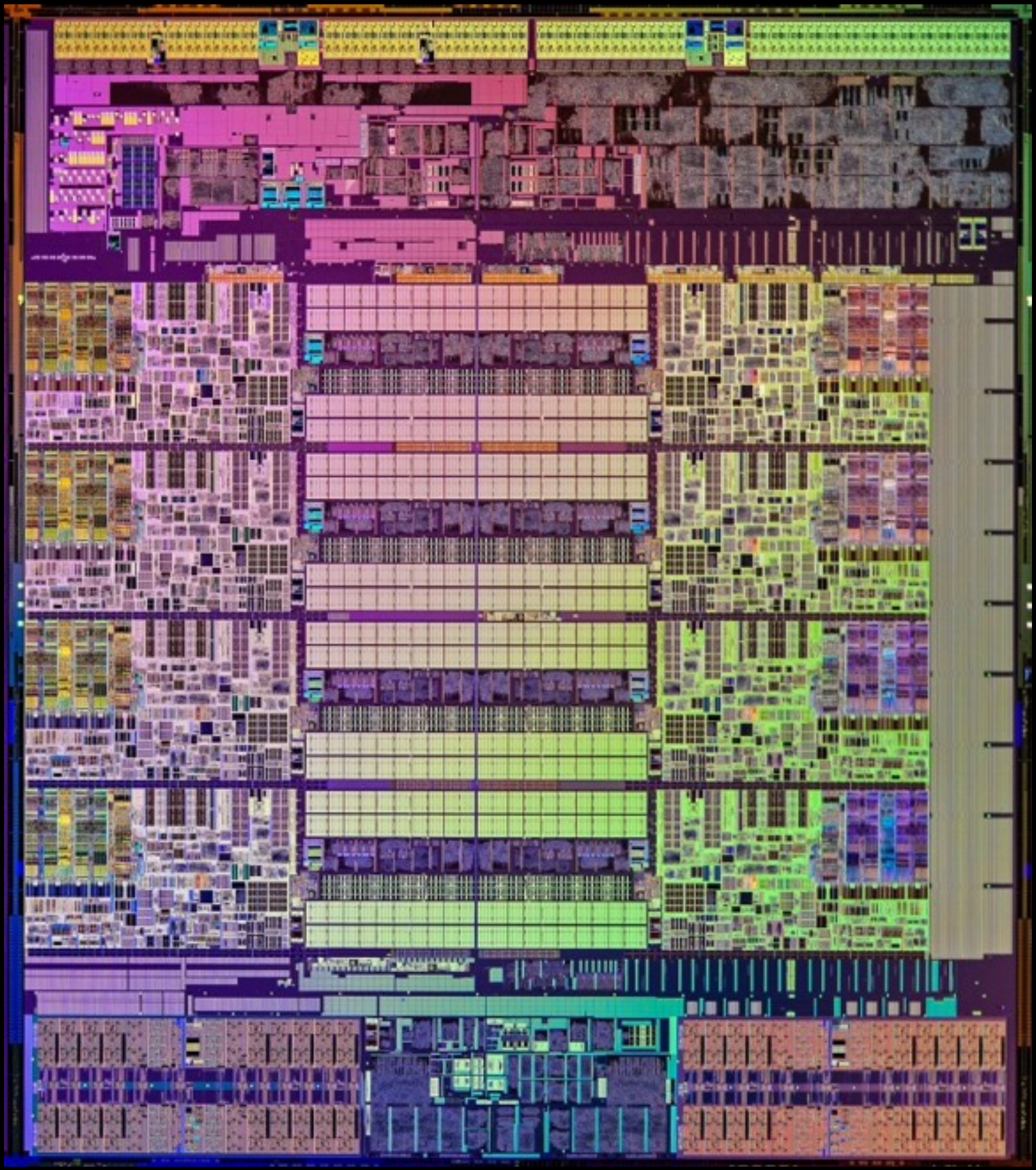
blippAR®

TOMTOM®

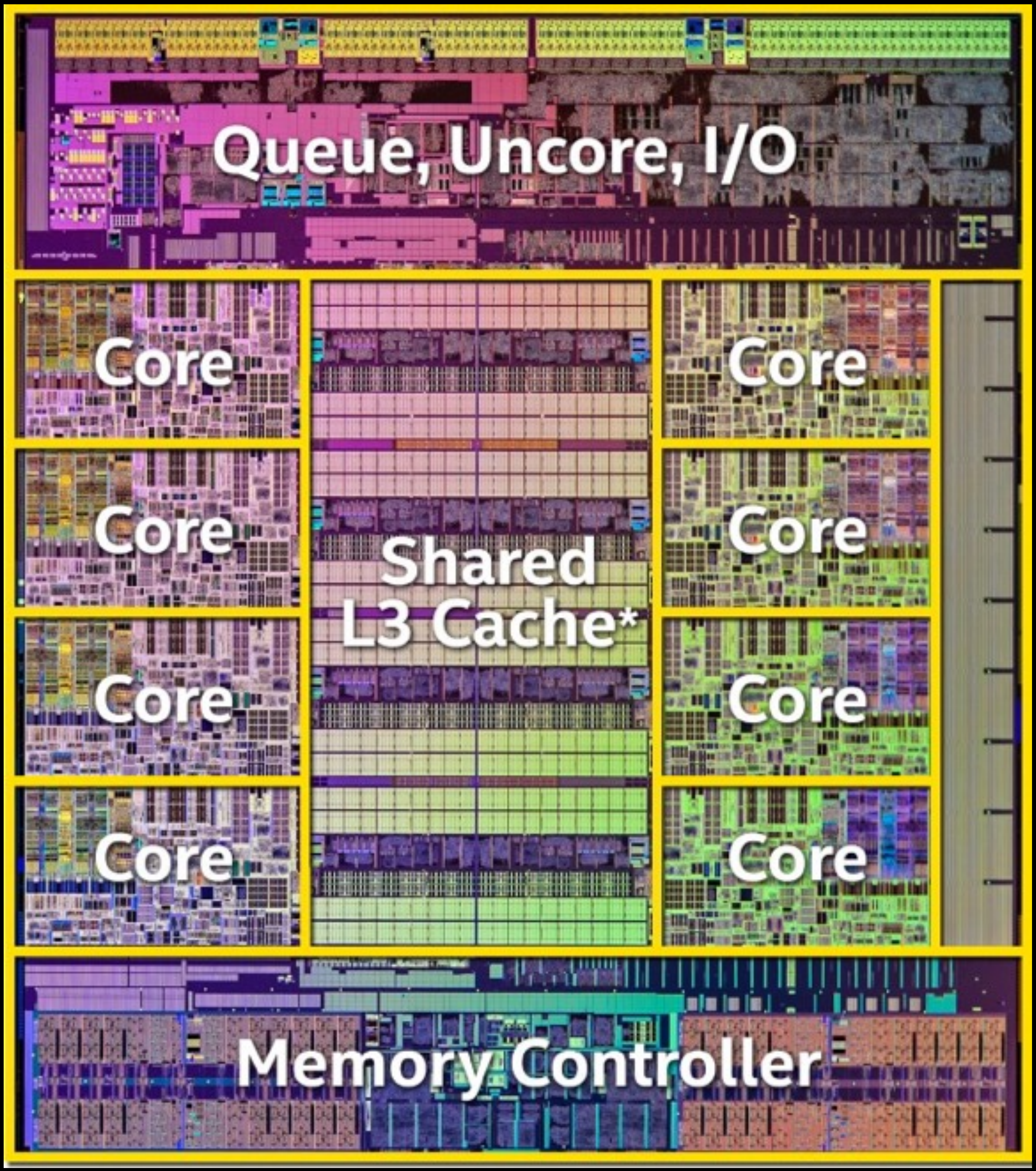
# Agenda

1. **Intro GPU architecture**
2. Intro GPU programming model
3. CUDA/OpenCL by example: matrix-multiplication
4. C++11 ❤️ GPU → SyCL

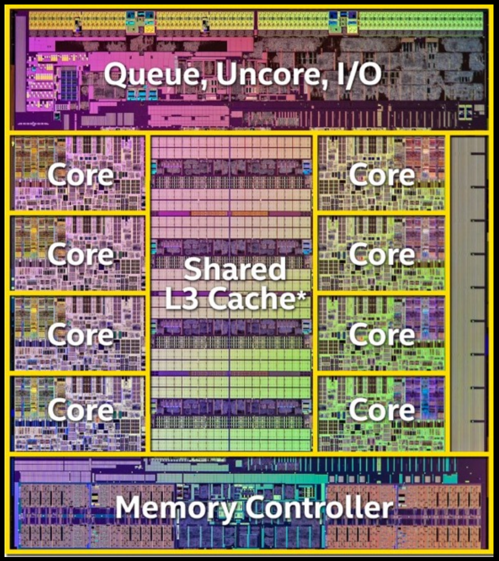
# Quiz: what is this?

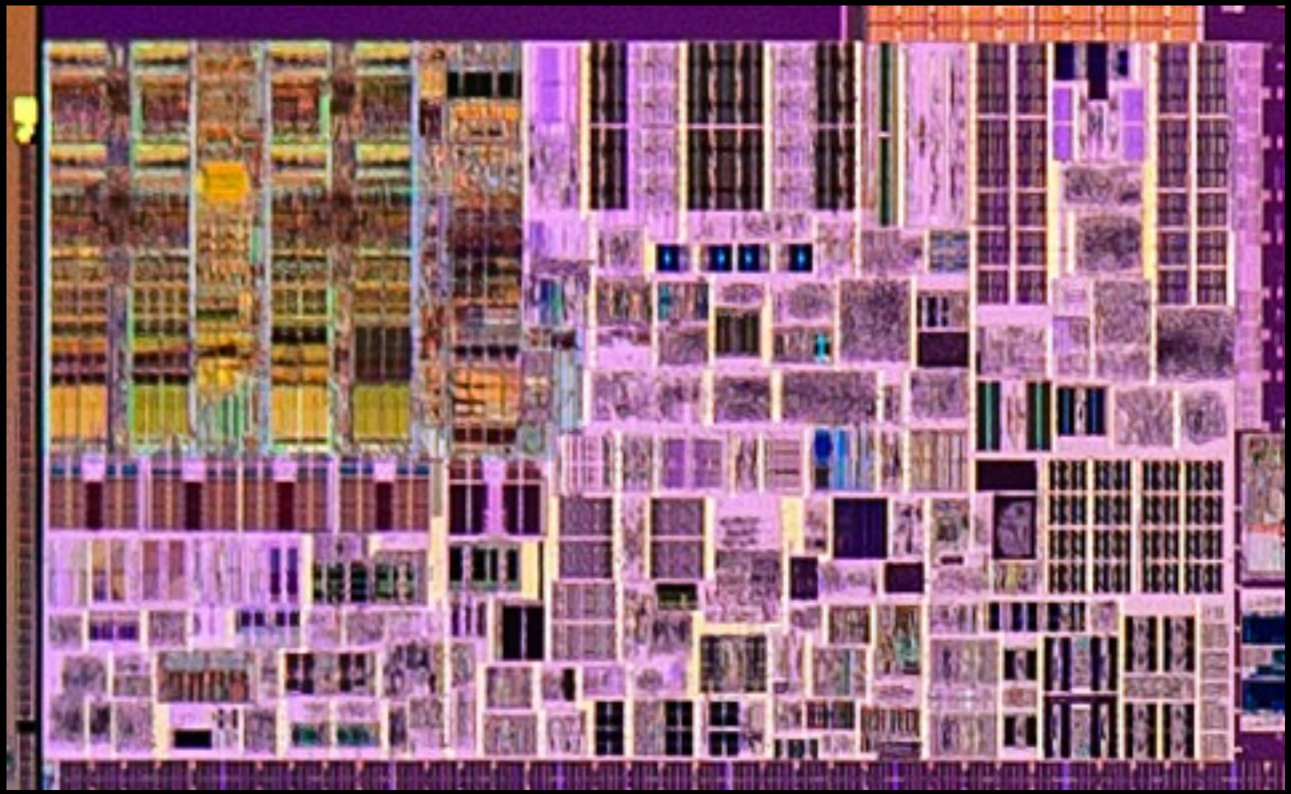


# Easier?


Queue, Uncore, I/O
Core
Core
Core
Core
Shared
L3 Cache*
Core
Core
Core
Core
Memory Controller

# But where is the ALU?



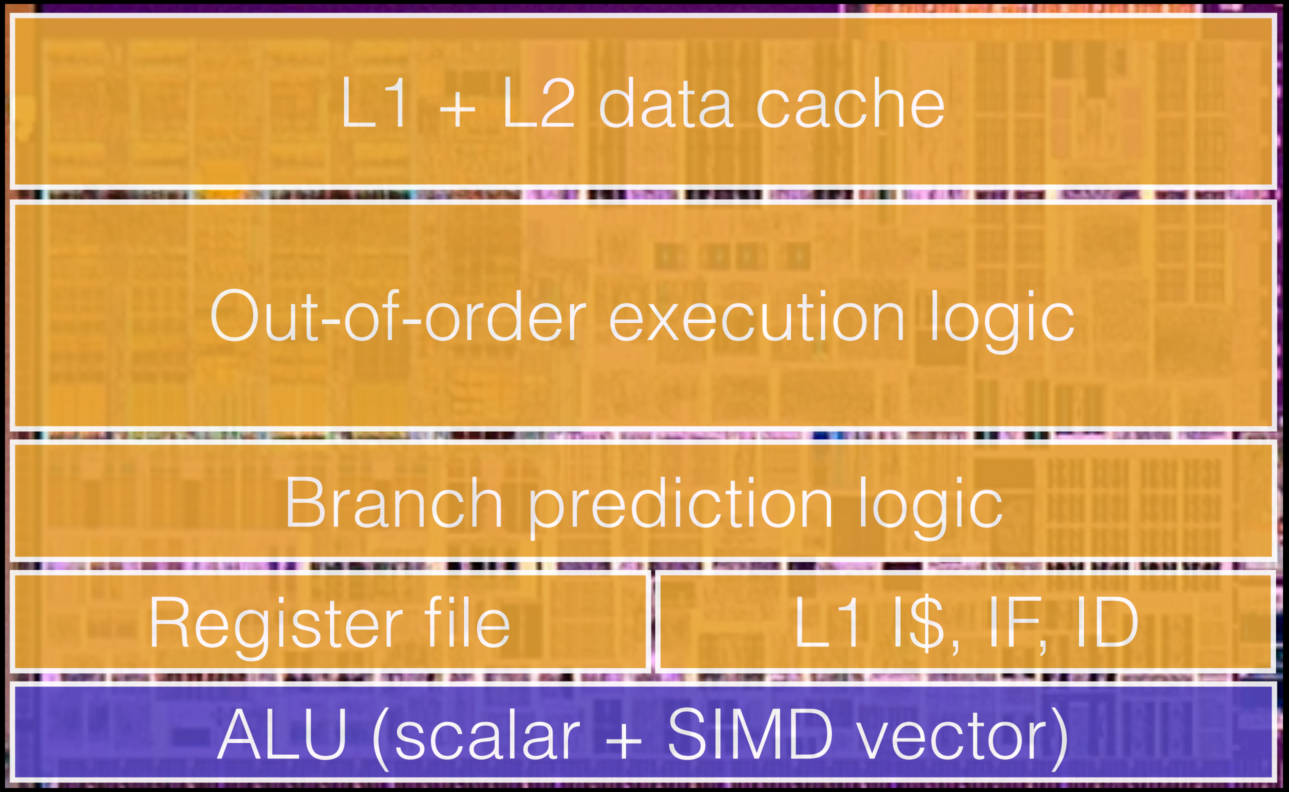


Haswell-E core

Total chip:
5,5 billion transistors

Simple 32-bit integer multiplier:
21 thousand transistors

# But where is the ALU?

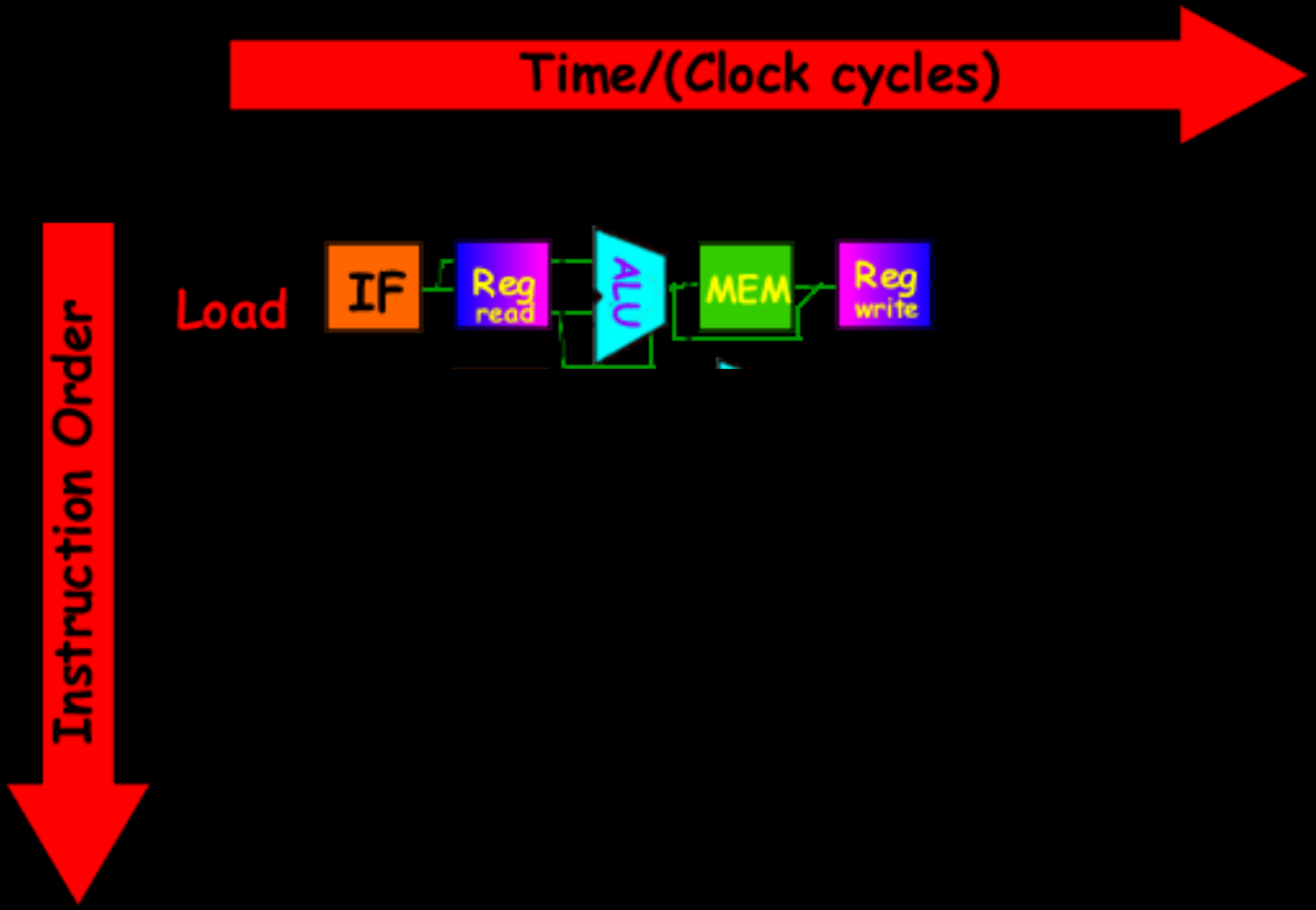


Haswell-E core

Lot’s of ‘useless logic’

Consequence:
only 8 multiplications
per clock-tick per core

# Why is the 'useless logic' needed?

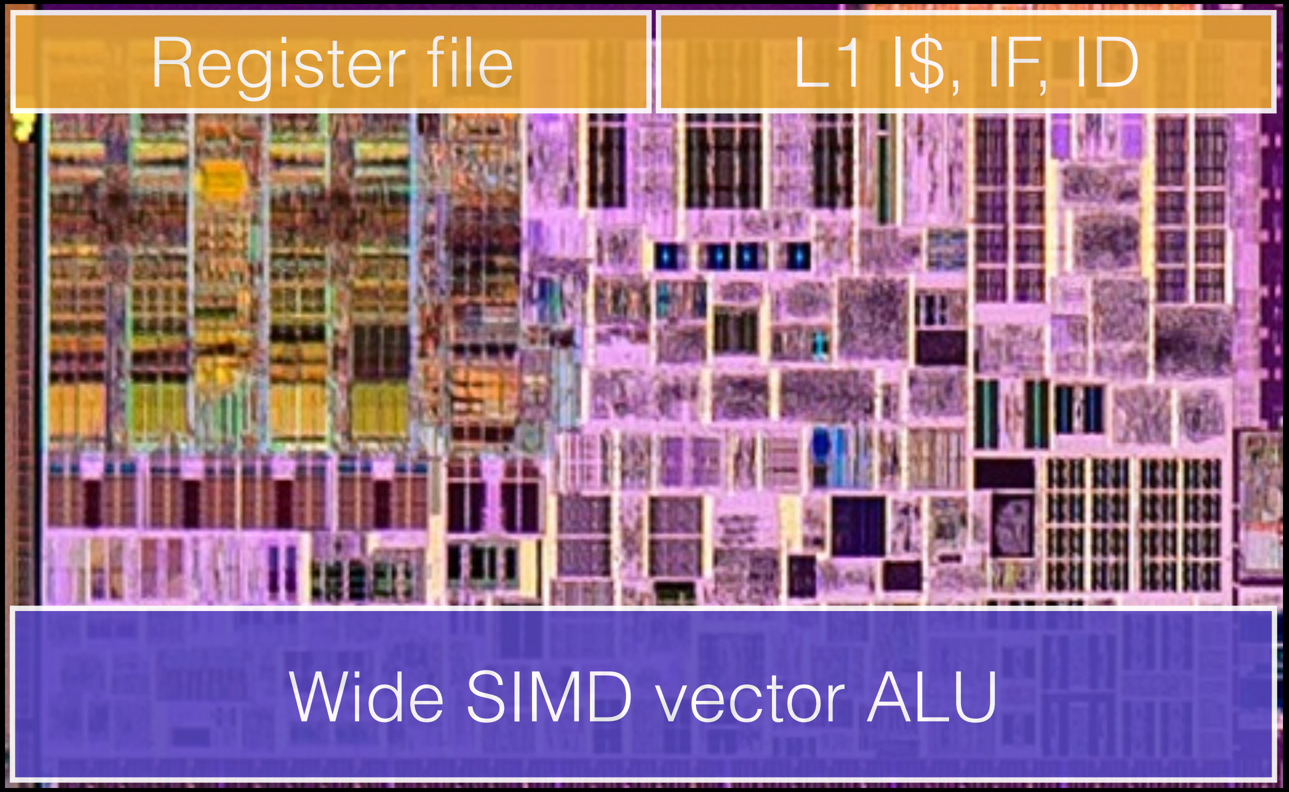


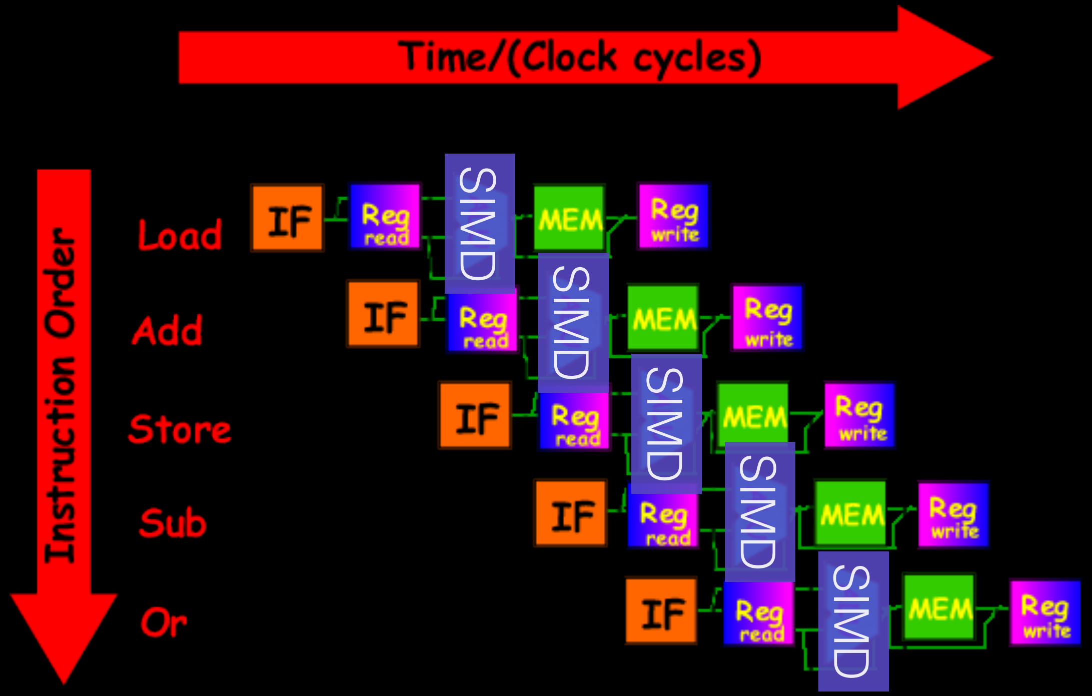
Haswell: 19 pipeline stages

Huge impact of branches & data dependencies, low ILP

# Now let's build a GPU

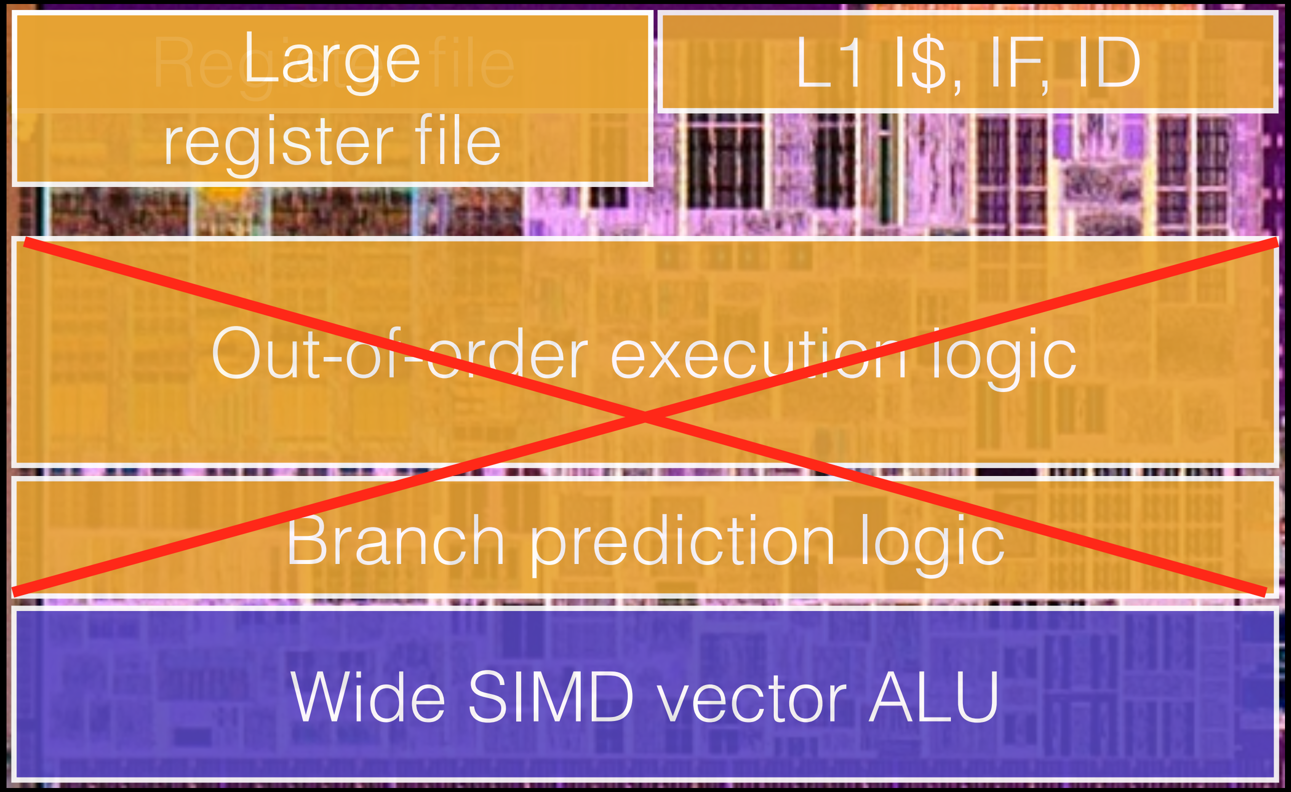
## Step 1: Vector ALU only





# Now let’s build a GPU

## Step 2: Multiple active threads



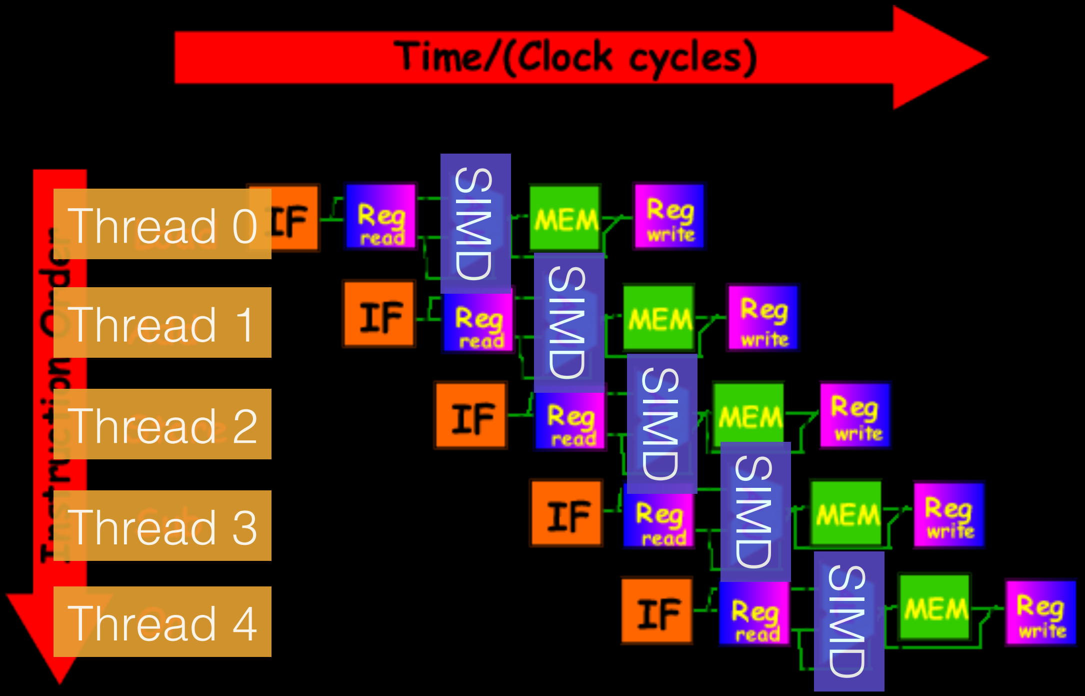
Large register file

L1 I$, IF, ID

~~Out-of-order execution logic~~

~~Branch prediction logic~~

Wide SIMD vector ALU



Time/(Clock cycles)

Thread 0

Thread 1

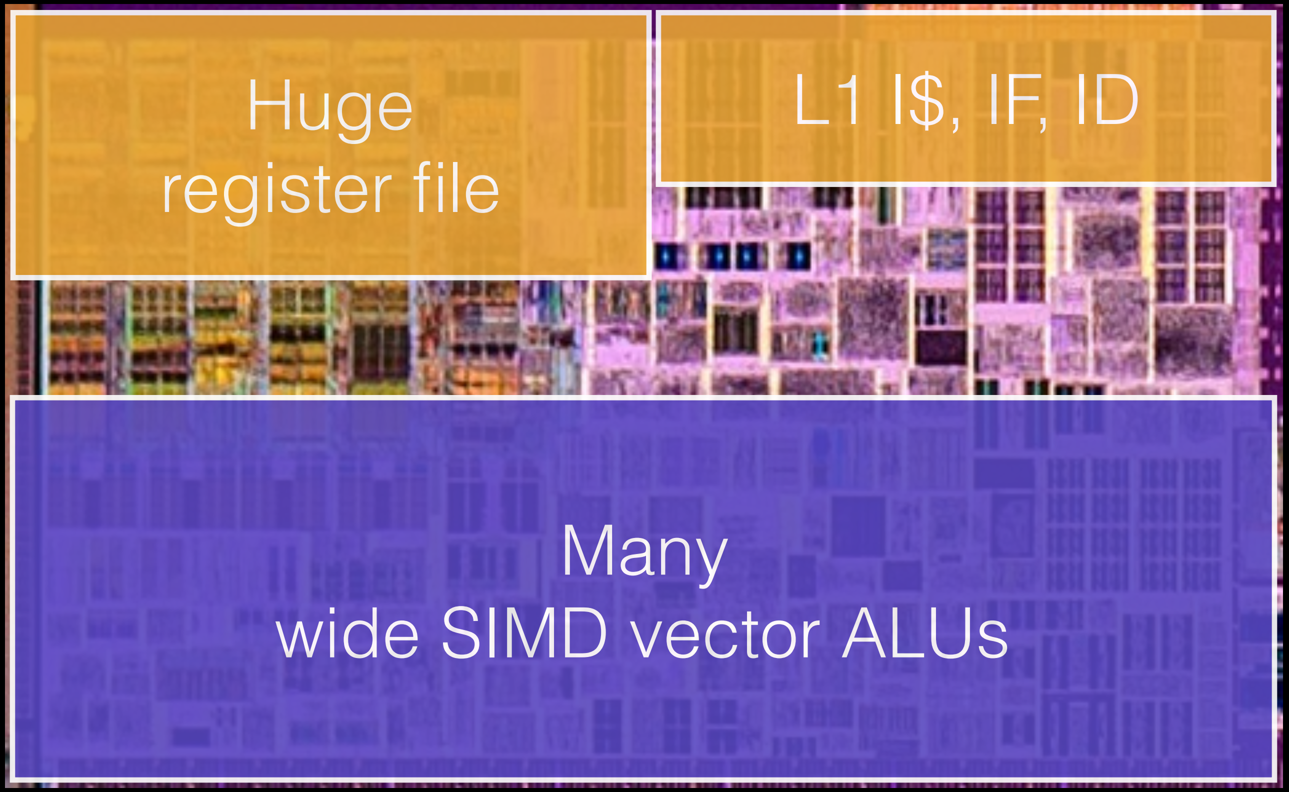
Thread 2

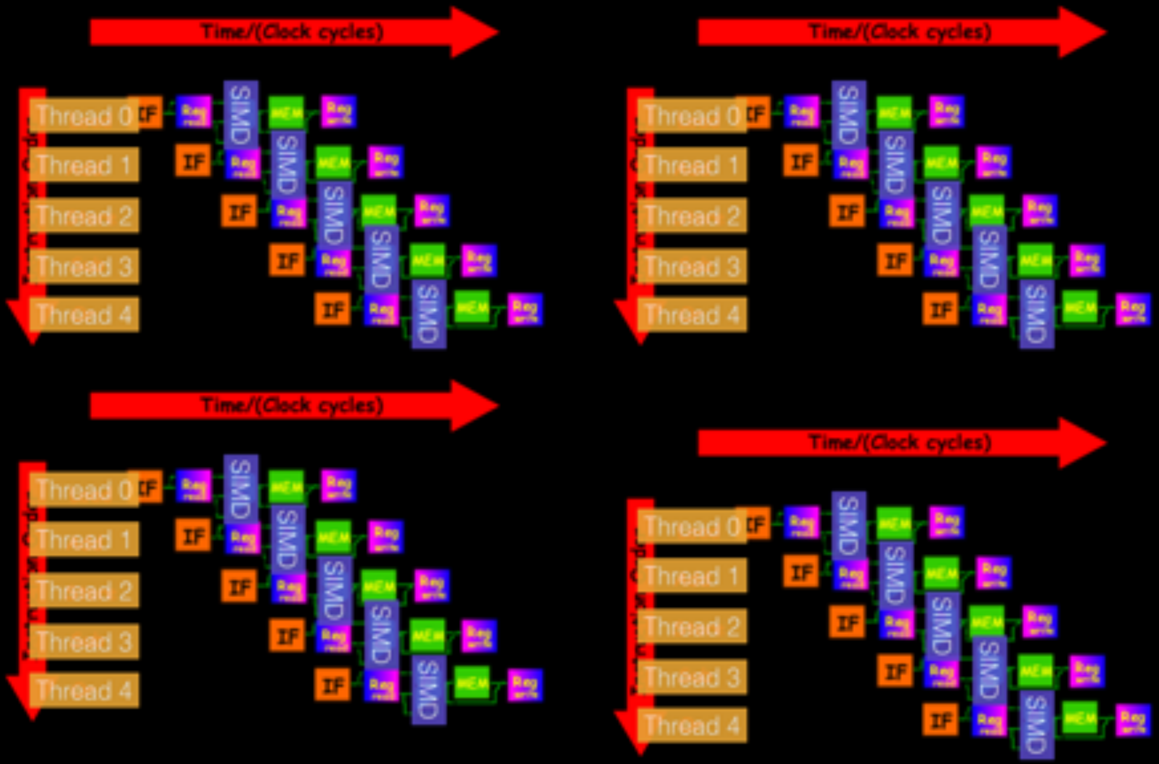
Thread 3

Thread 4

# Now let's build a GPU

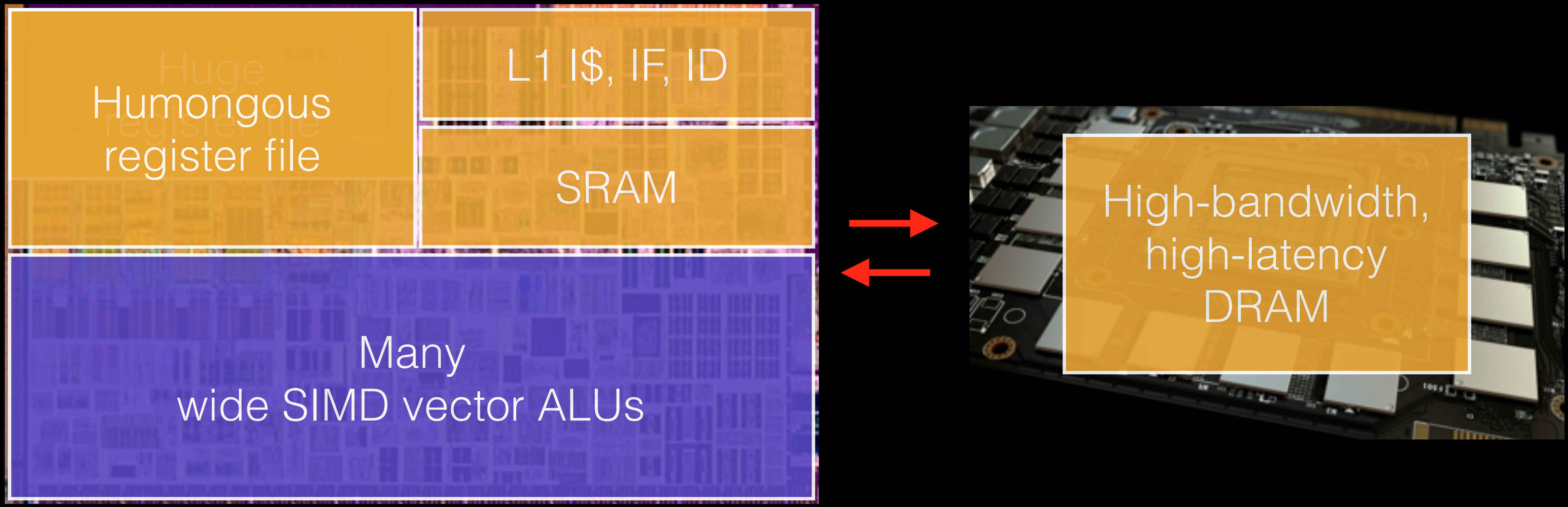
## Step 3: More more more!





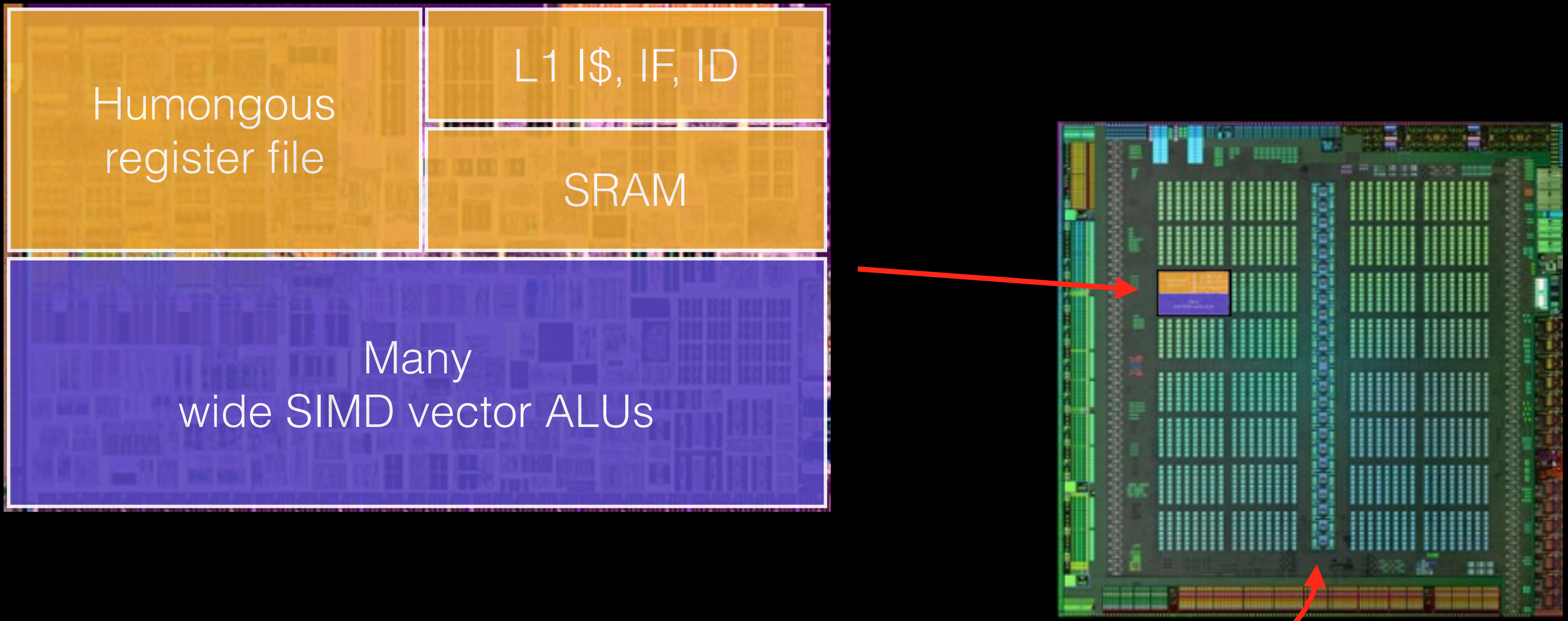
# Now let’s build a GPU

## Step 4: High-bandwidth off-chip memory



# Now let's build a GPU

## Step 5: Duplicate this 'core' / 'SM'



Humongous register file

L1 I$, IF, ID

SRAM

Many
wide SIMD vector ALUs

L2 cache

# GPU vs CPU

CPU

130 pJ/flop (SIMD SP)

Optimised for latency

Low latency DRAM

Deep cache hierarchy

Few active threads

GPU

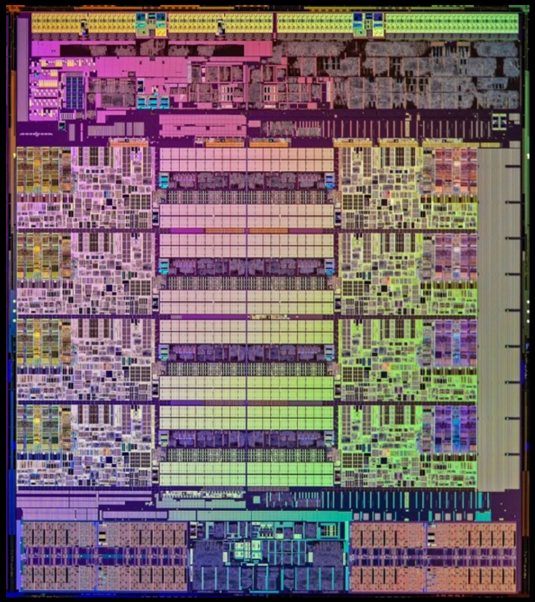
30 pJ/flop (SP)

Optimised for throughput

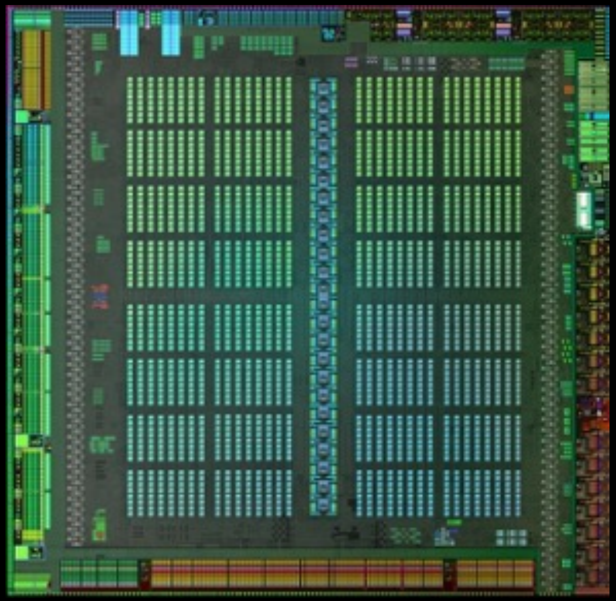
High bandwidth DRAM

Explicit management of SRAM

Many active threads



Haswell-E



Maxwell

# Agenda

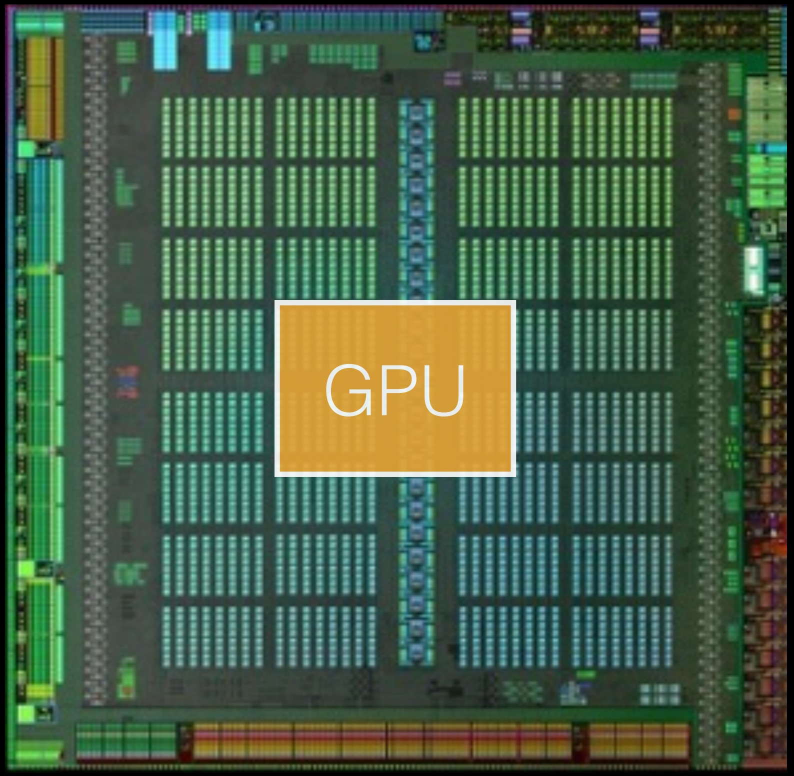
1. Intro GPU architecture

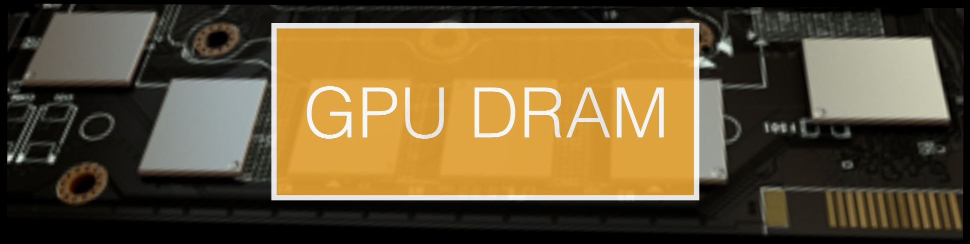
2. **Intro GPU programming model**

3. CUDA/OpenCL by example: matrix-multiplication

4. C++11 ❤️ GPU → SyCL

# So, let’s program this GPU!

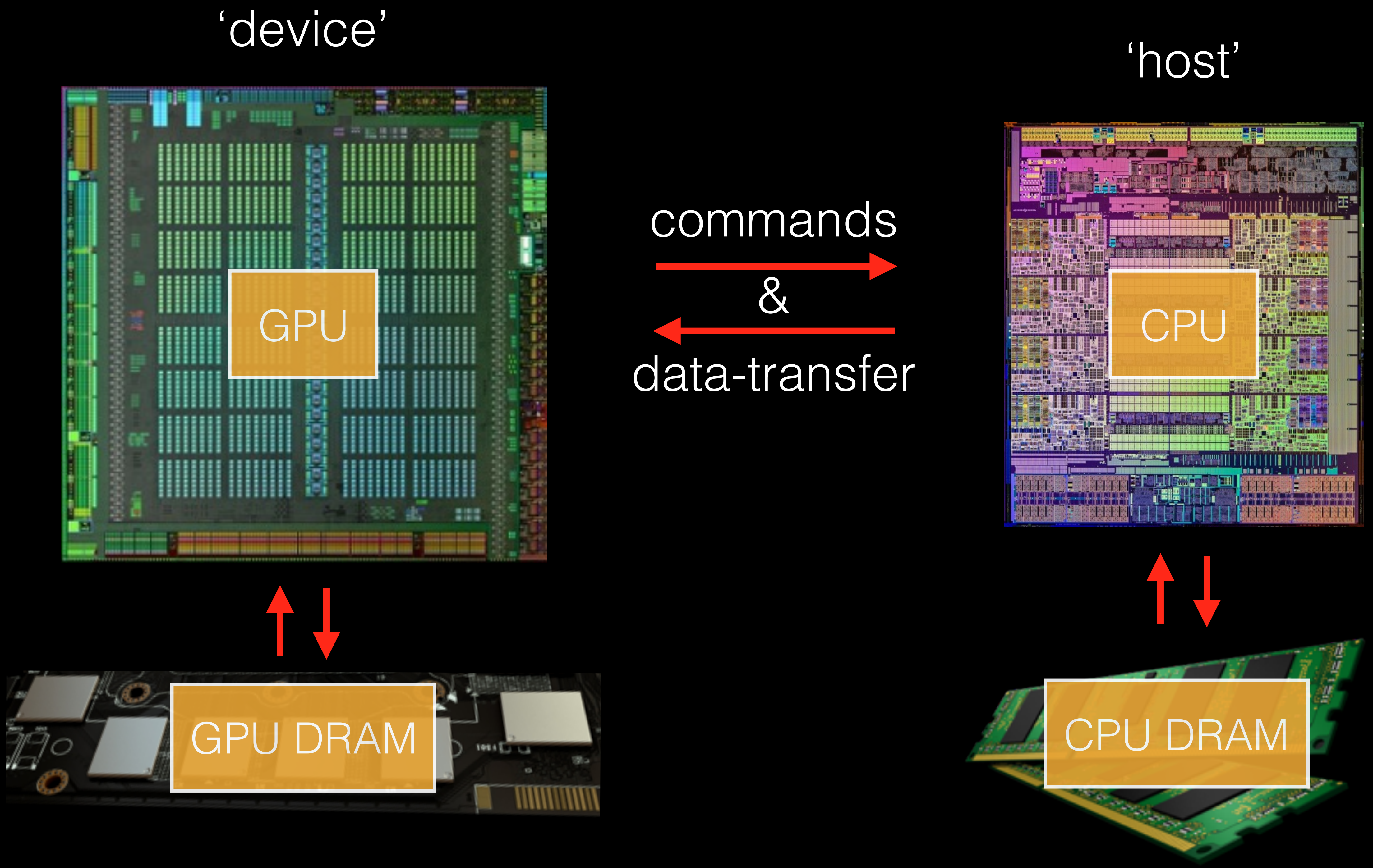




But, how?

1. No access to the file system
2. No I/O, no `printf()` or `std::cout`
3. No operating system
4. Separate memory

# The CPU as a control processor



'device'

GPU

GPU DRAM

commands

&

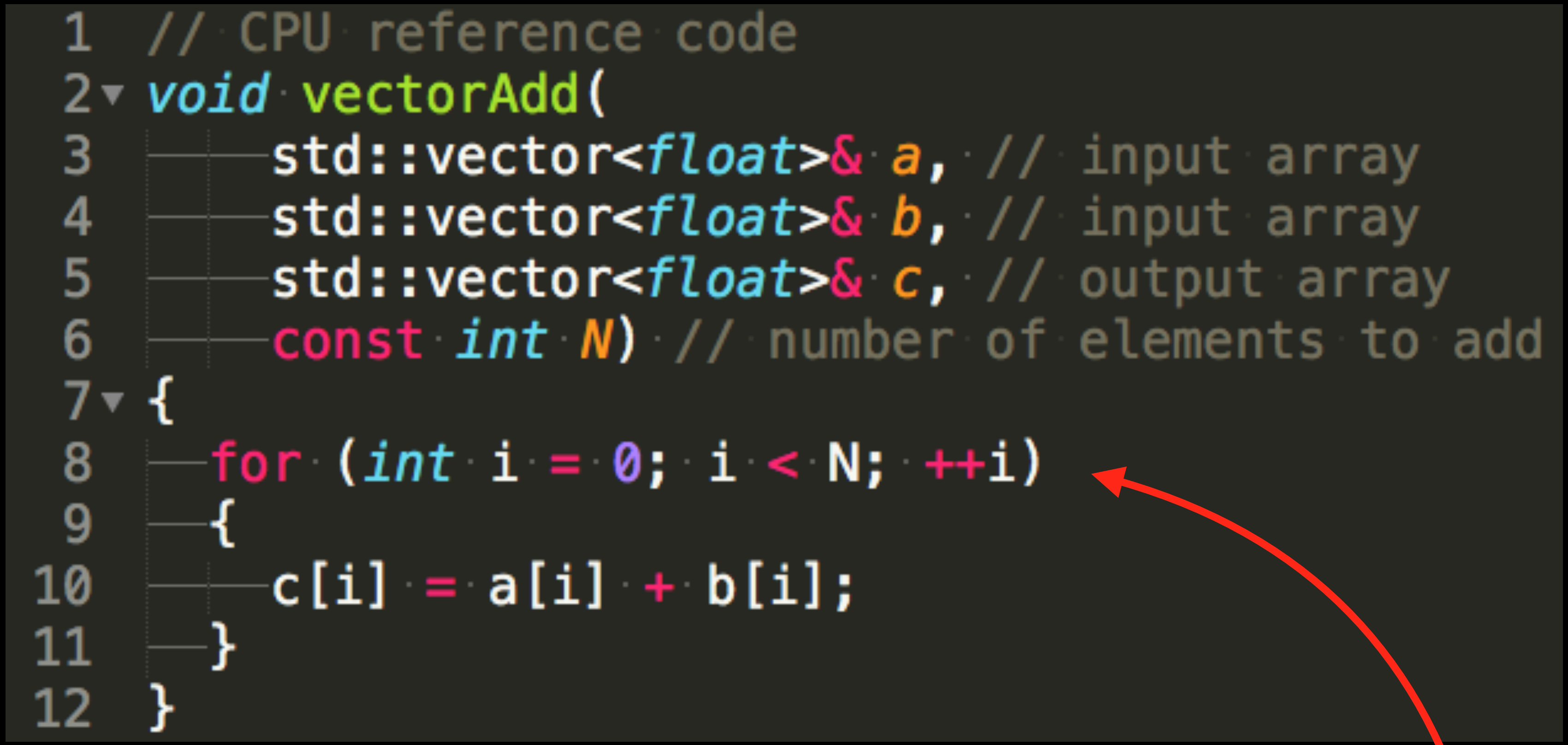
data-transfer

'host'

CPU

CPU DRAM

# Vector add is the new ‘hello world’



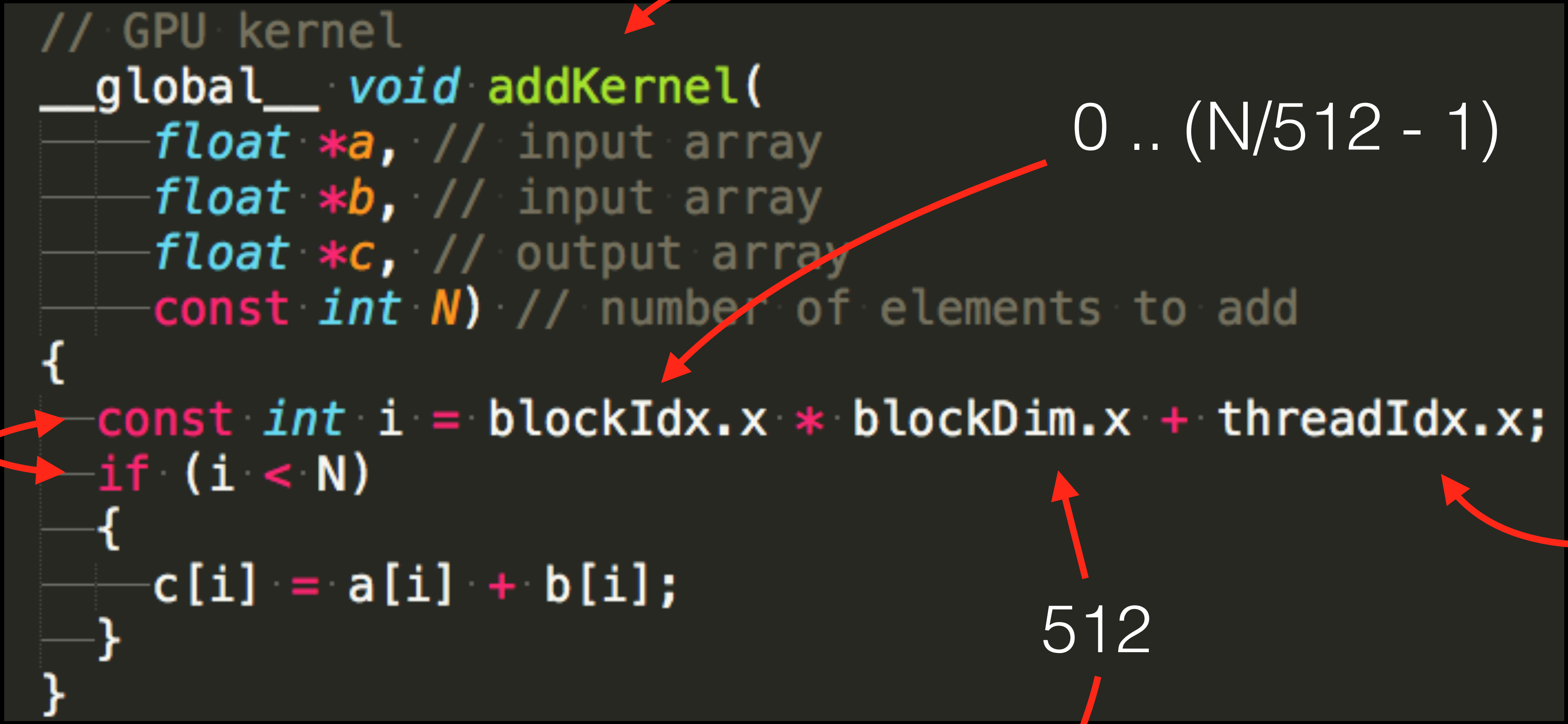
```cpp
// CPU reference code
void vectorAdd(
        std::vector<float>& a, // input array
        std::vector<float>& b, // input array
        std::vector<float>& c, // output array
        const int N) // number of elements to add
{
    for (int i = 0; i < N; ++i)
    {
        c[i] = a[i] + b[i];
    }
}
```

Candidate for parallelisation

# The kernel

For incomplete blocks

‘Function’ to be executed by each thread



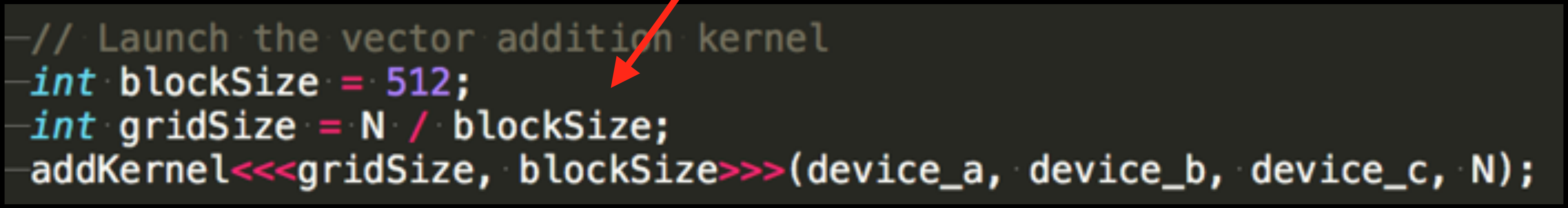
```
// GPU kernel
__global__ void addKernel(
        float *a, // input array
        float *b, // input array
        float *c, // output array
        const int N) // number of elements to add
{
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
    {
        c[i] = a[i] + b[i];
    }
}
```

0 .. (N/512 - 1)

0 .. 511

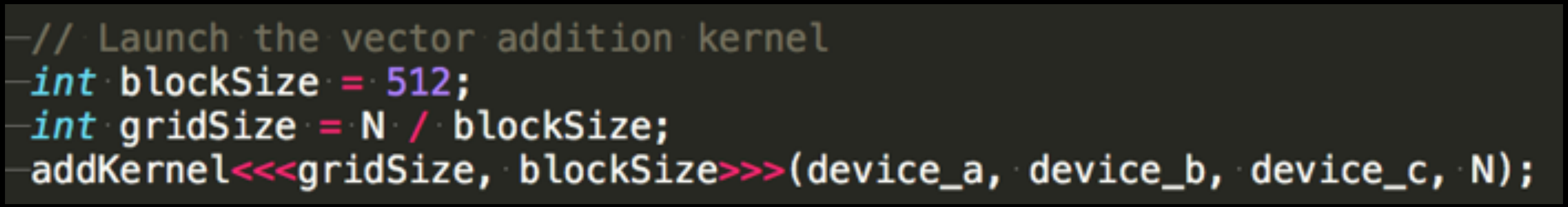
512

Replacement of the for-loop index

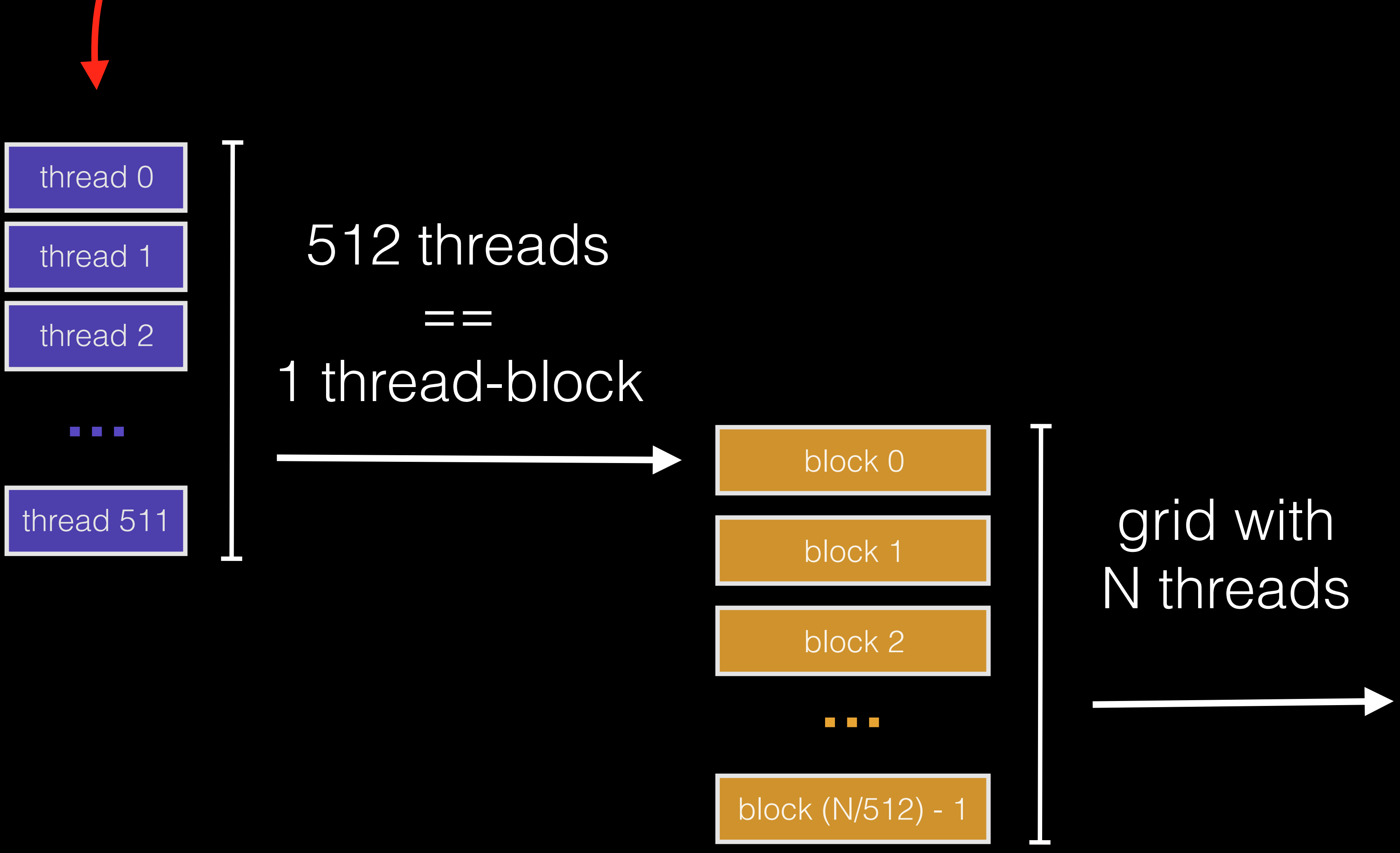


```
// Launch the vector addition kernel
int blockSize = 512;
int gridSize = N / blockSize;
addKernel<<<gridSize, blockSize>>>(device_a, device_b, device_c, N);
```

# Threads and thread-blocks



```
// Launch the vector addition kernel
int blockSize = 512;
int gridSize = N / blockSize;
addKernel<<<gridSize, blockSize>>>(device_a, device_b, device_c, N);
```



thread 0

thread 1

thread 2

...

thread 511

512 threads
==
1 thread-block

block 0

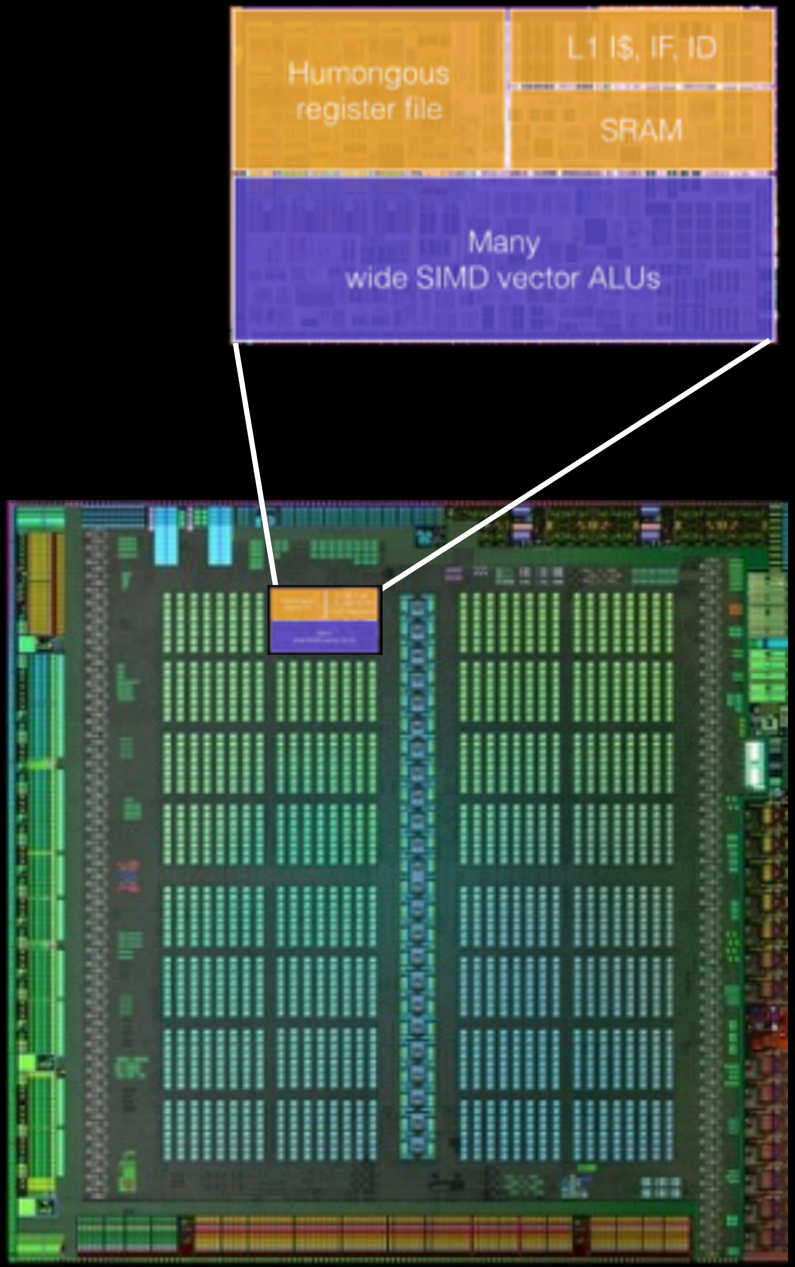
block 1

block 2

...

block (N/512) - 1

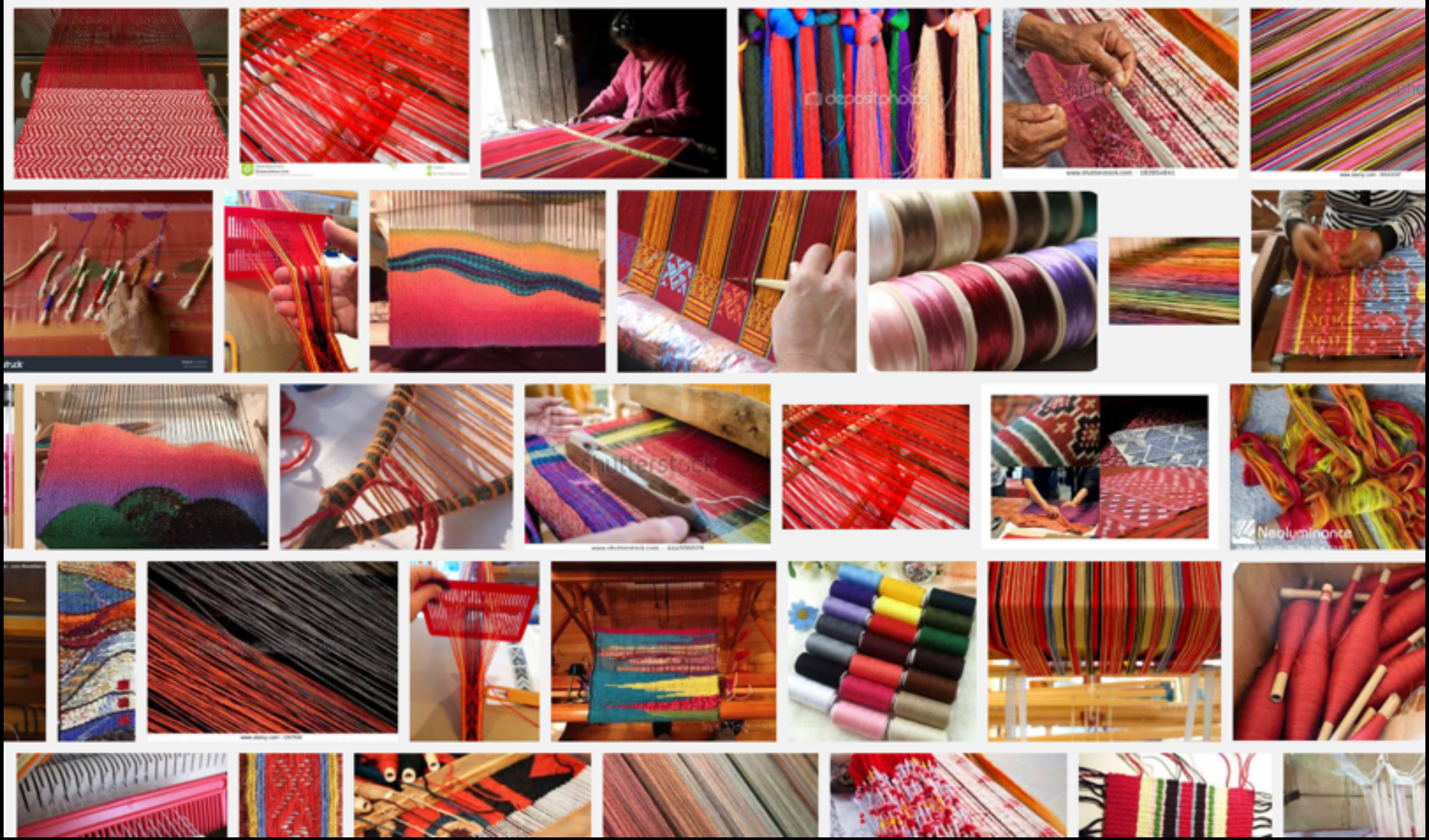
grid with
N threads



Note:

thread-blocks and grids can also be 2D or 3D

# Weaving threads



# Weaving threads (aka scheduling)

Scheduling unit: **thread-block**

Multiple blocks per SM, but:

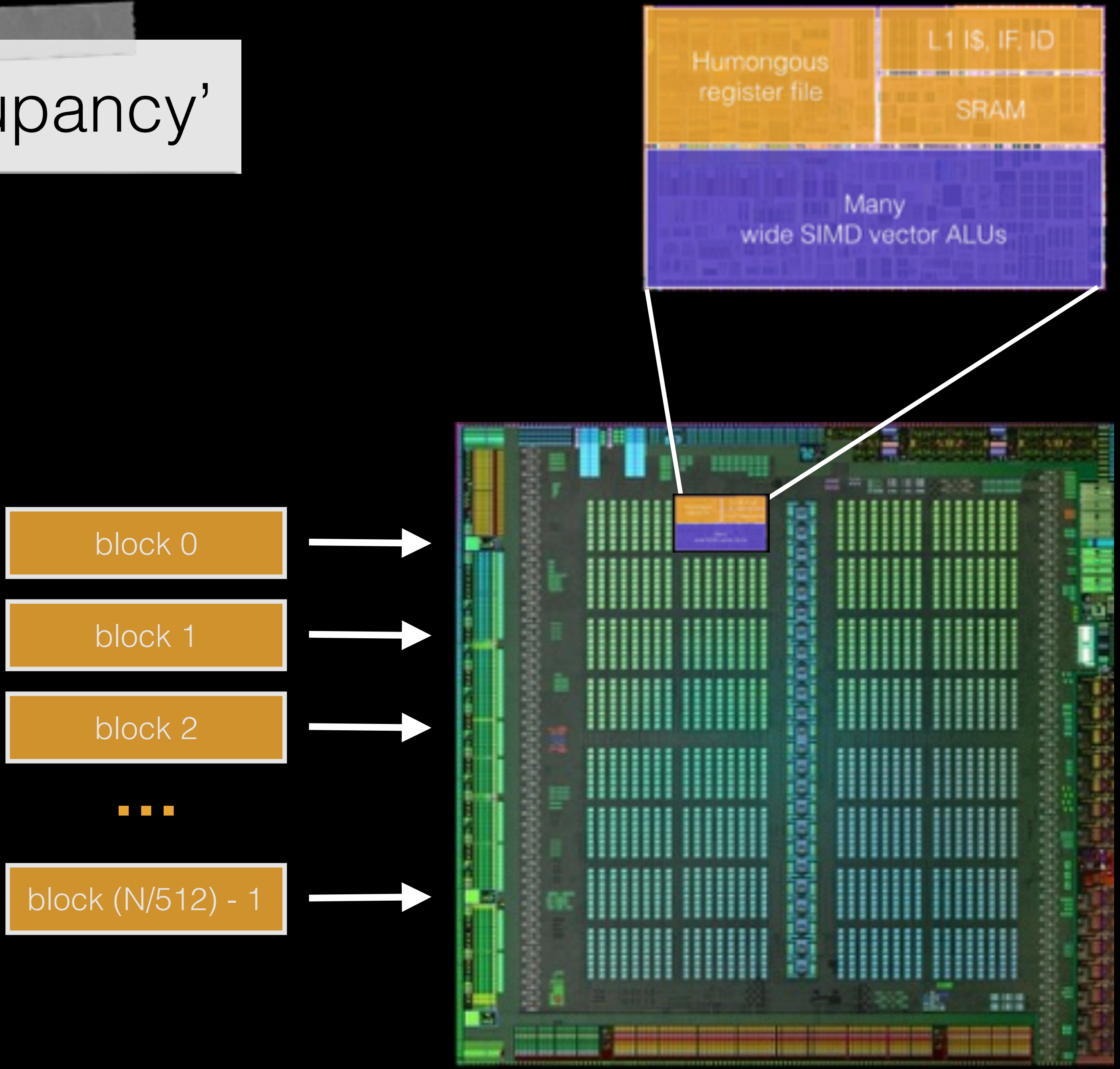
1. Maximum 2048 threads
2. Maximum 32 blocks
3. Maximum 64K registers
4. Maximum 64KB shared memory

'occupancy'

Why do we care?

1. Share SRAM 'shared memory'
2. Synchronisation barriers

Otherwise: **no synchronisation!**

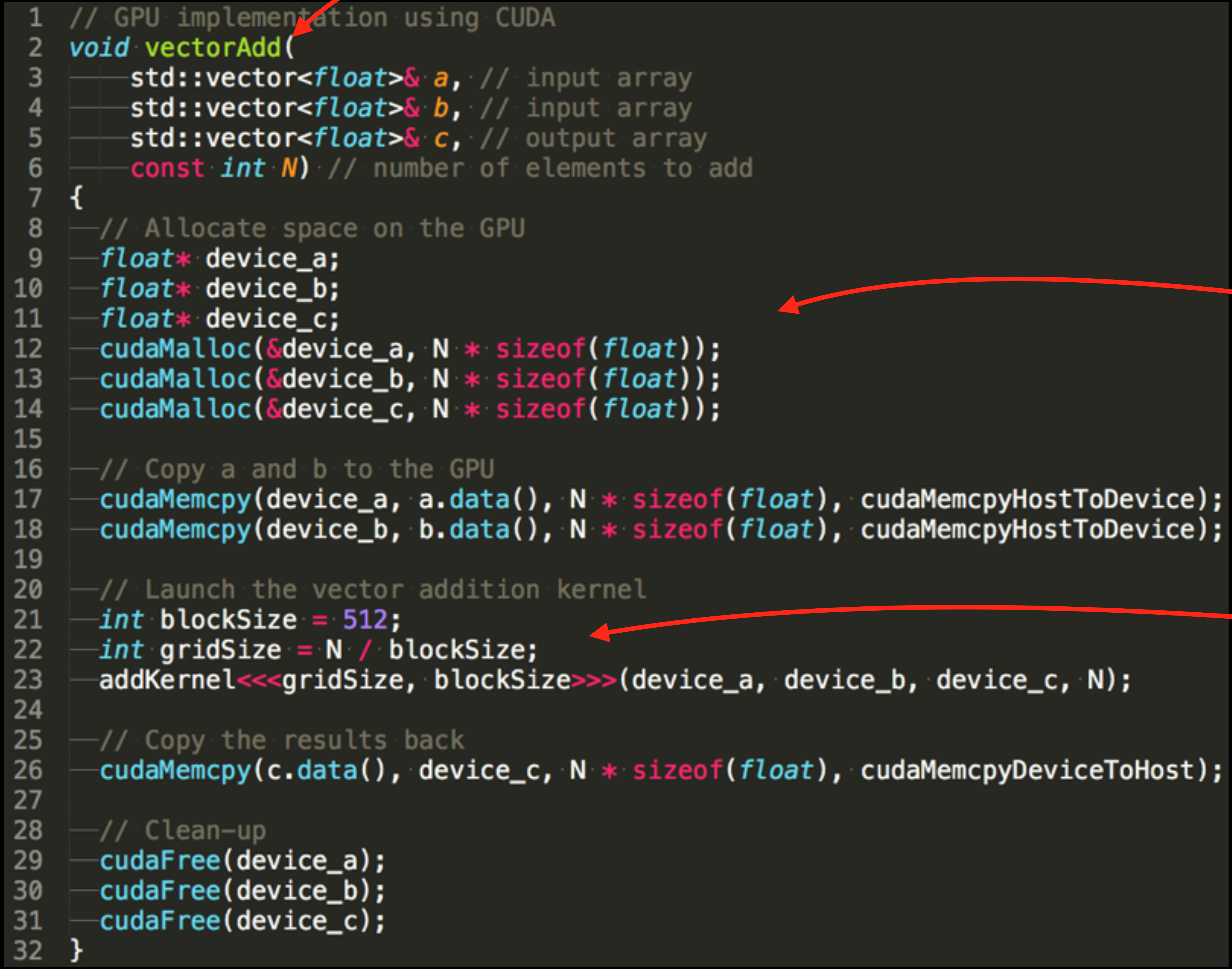


Note:

lower level scheduling: threads execute in 'warps' of 32 on SIMD units

# The boilerplate code

CPU function

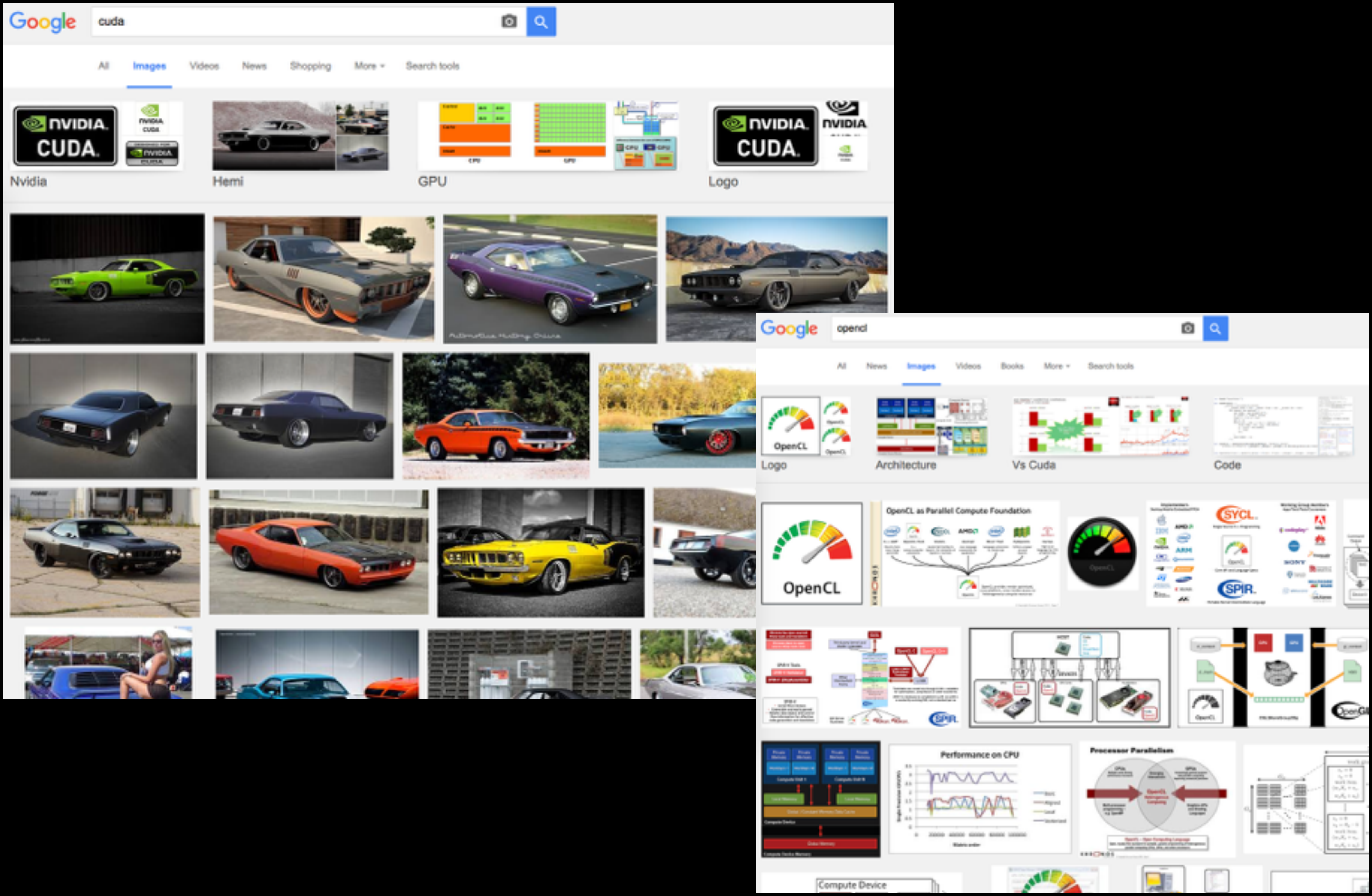


```
// GPU implementation using CUDA
void vectorAdd(
    std::vector<float>& a, // input array
    std::vector<float>& b, // input array
    std::vector<float>& c, // output array
    const int N) // number of elements to add
{
  // Allocate space on the GPU
  float* device_a;
  float* device_b;
  float* device_c;
  cudaMalloc(&device_a, N * sizeof(float));
  cudaMalloc(&device_b, N * sizeof(float));
  cudaMalloc(&device_c, N * sizeof(float));

  // Copy a and b to the GPU
  cudaMemcpy(device_a, a.data(), N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(device_b, b.data(), N * sizeof(float), cudaMemcpyHostToDevice);

  // Launch the vector addition kernel
  int blockSize = 512;
  int gridSize = N / blockSize;
  addKernel<<<gridSize, blockSize>>>(device_a, device_b, device_c, N);

  // Copy the results back
  cudaMemcpy(c.data(), device_c, N * sizeof(float), cudaMemcpyDeviceToHost);

  // Clean-up
  cudaFree(device_a);
  cudaFree(device_b);
  cudaFree(device_c);
}
```

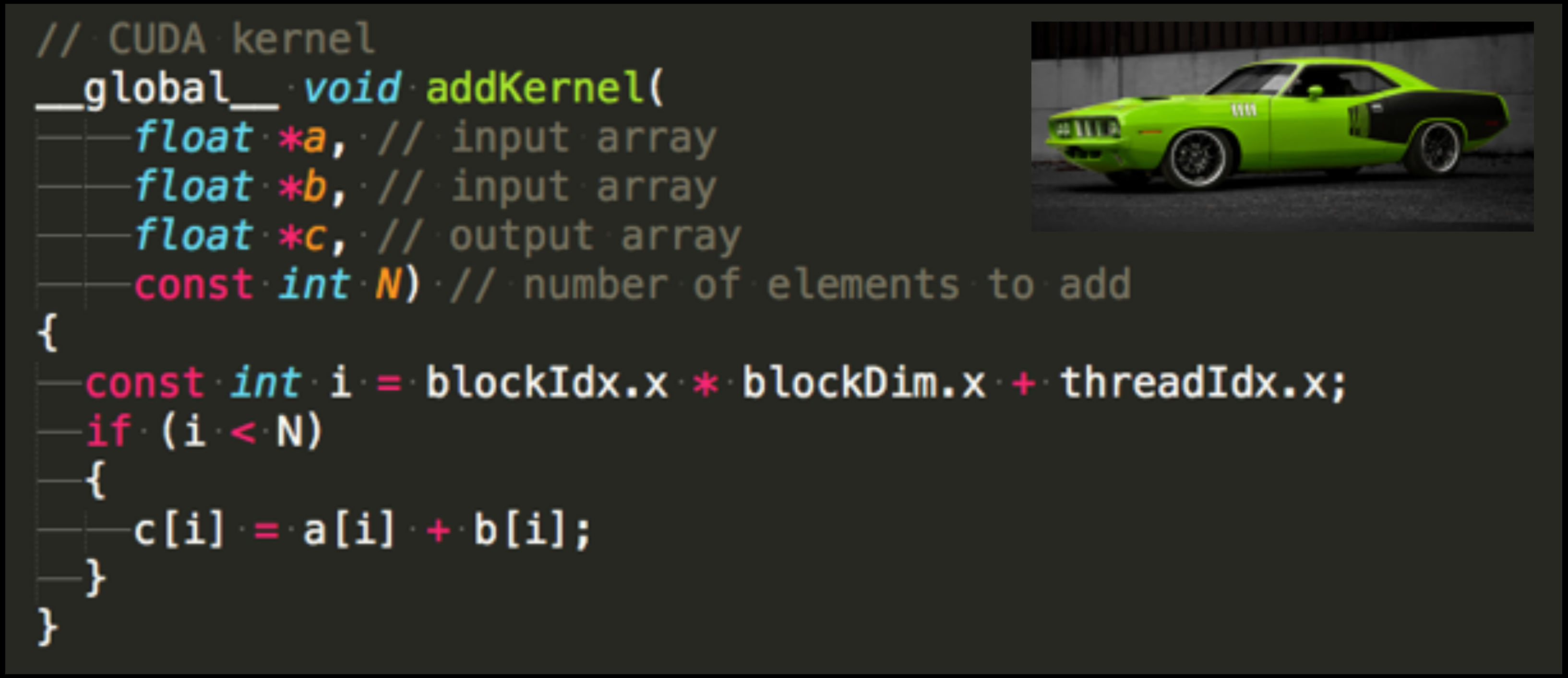
Pointers to GPU DRAM

N threads

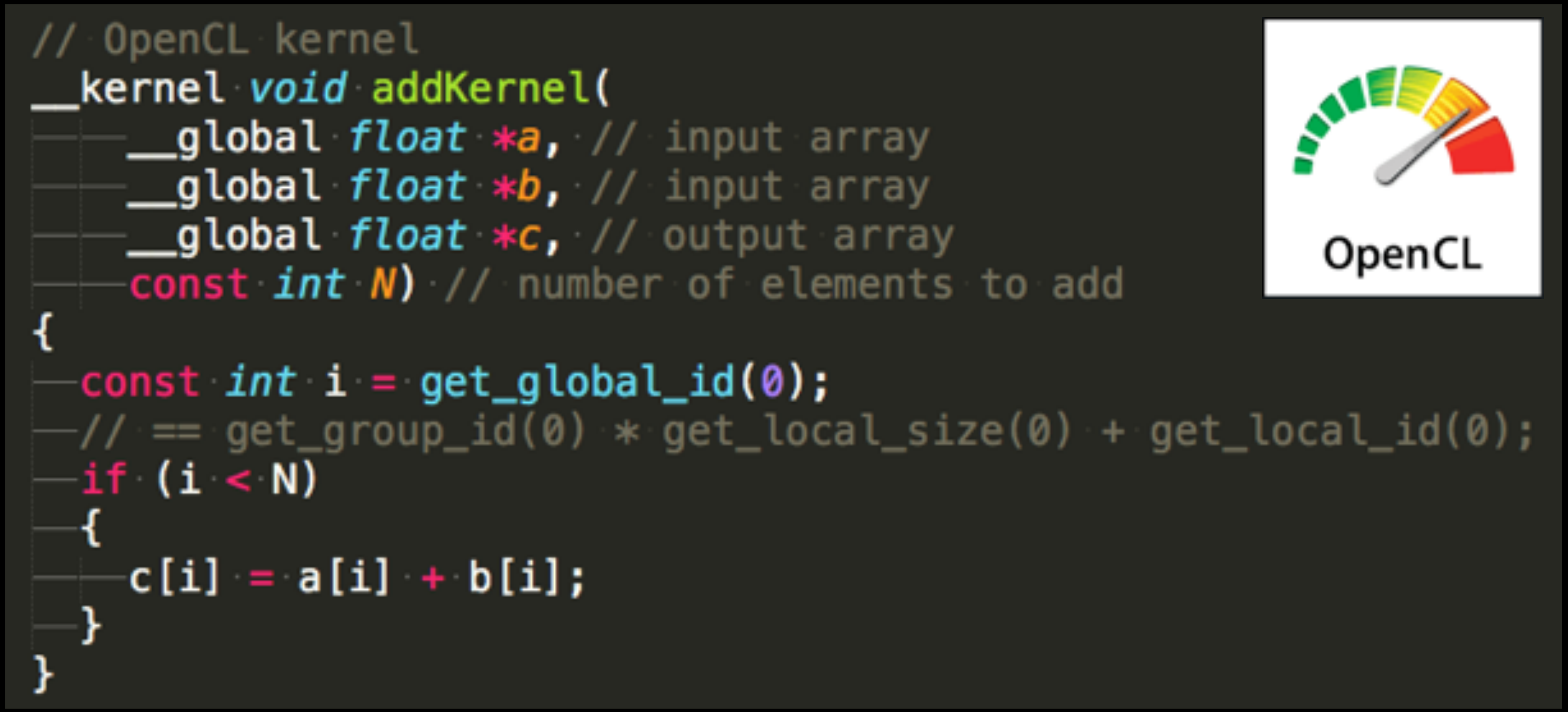
# CUDA versus OpenCL



# CUDA versus OpenCL



```
// CUDA kernel
__global__ void addKernel(
		float *a, // input array
		float *b, // input array
		float *c, // output array
		const int N) // number of elements to add
{
	const int i = blockIdx.x * blockDim.x + threadIdx.x;
	if (i < N)
	{
		c[i] = a[i] + b[i];
	}
}
```



```
// OpenCL kernel
__kernel void addKernel(
		__global float *a, // input array
		__global float *b, // input array
		__global float *c, // output array
		const int N) // number of elements to add
{
	const int i = get_global_id(0);
	// == get_group_id(0) * get_local_size(0) + get_local_id(0);
	if (i < N)
	{
		c[i] = a[i] + b[i];
	}
}
```

OpenCL

**The good:**

Kernels are almost the same

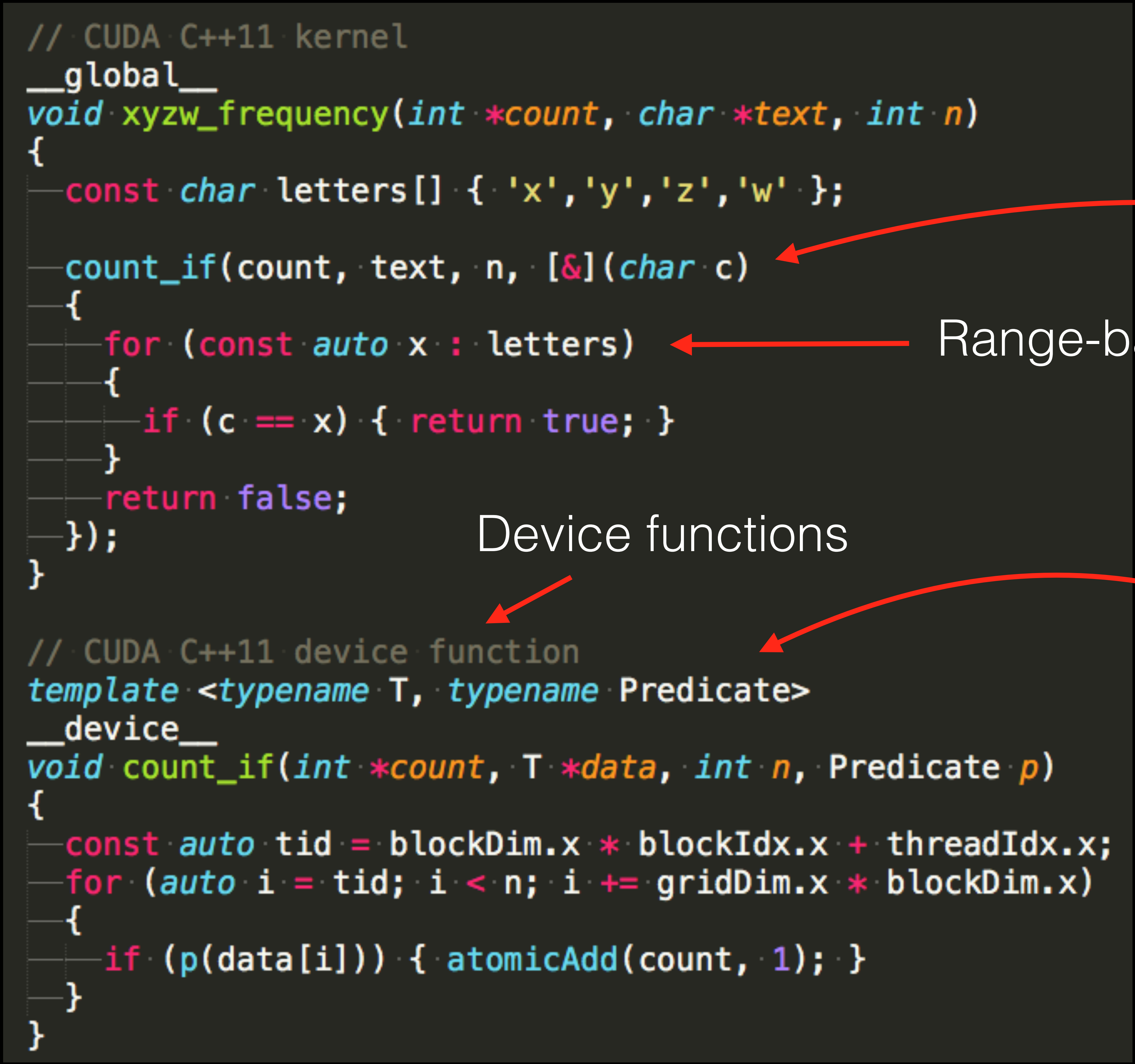
**The bad:**

OpenCL host code is much more verbose

**The ugly:**

Performance portability is far from trivial

# What about (modern) C++?



```
// CUDA C++11 kernel
__global__
void xyzw_frequency(int *count, char *text, int n)
{
  const char letters[] { 'x','y','z','w' };

  count_if(count, text, n, [&](char c)
  {
    for (const auto x : letters)
    {
      if (c == x) { return true; }
    }
    return false;
  });
}

// CUDA C++11 device function
template <typename T, typename Predicate>
__device__
void count_if(int *count, T *data, int n, Predicate p)
{
  const auto tid = blockDim.x * blockIdx.x + threadIdx.x;
  for (auto i = tid; i < n; i += gridDim.x * blockDim.x)
  {
    if (p(data[i])) { atomicAdd(count, 1); }
  }
}
```

Lambda’s

Range-based for-loops

Device functions

Templates

Auto type deduction

# What about (modern) C++ in kernels?

| | (subset of)<br>C | (subset of)<br>C++ | (subset of)<br>C++11/14 |
|---|---|---|---|
| CUDA < 7.0 | ✔ | ✔ | X |
| CUDA ≥ 7.0 | ✔ | ✔ | ✔ |
| OpenCL < 2.1 | ✔ | X | X |
| OpenCL ≥ 2.1 | ✔ | ✔ | ✔ |

No implementation yet

# Agenda

1. Intro GPU architecture
2. Intro GPU programming model
3. **CUDA/OpenCL by example: matrix-multiplication**
4. C++11 ❤️ GPU → SyCL

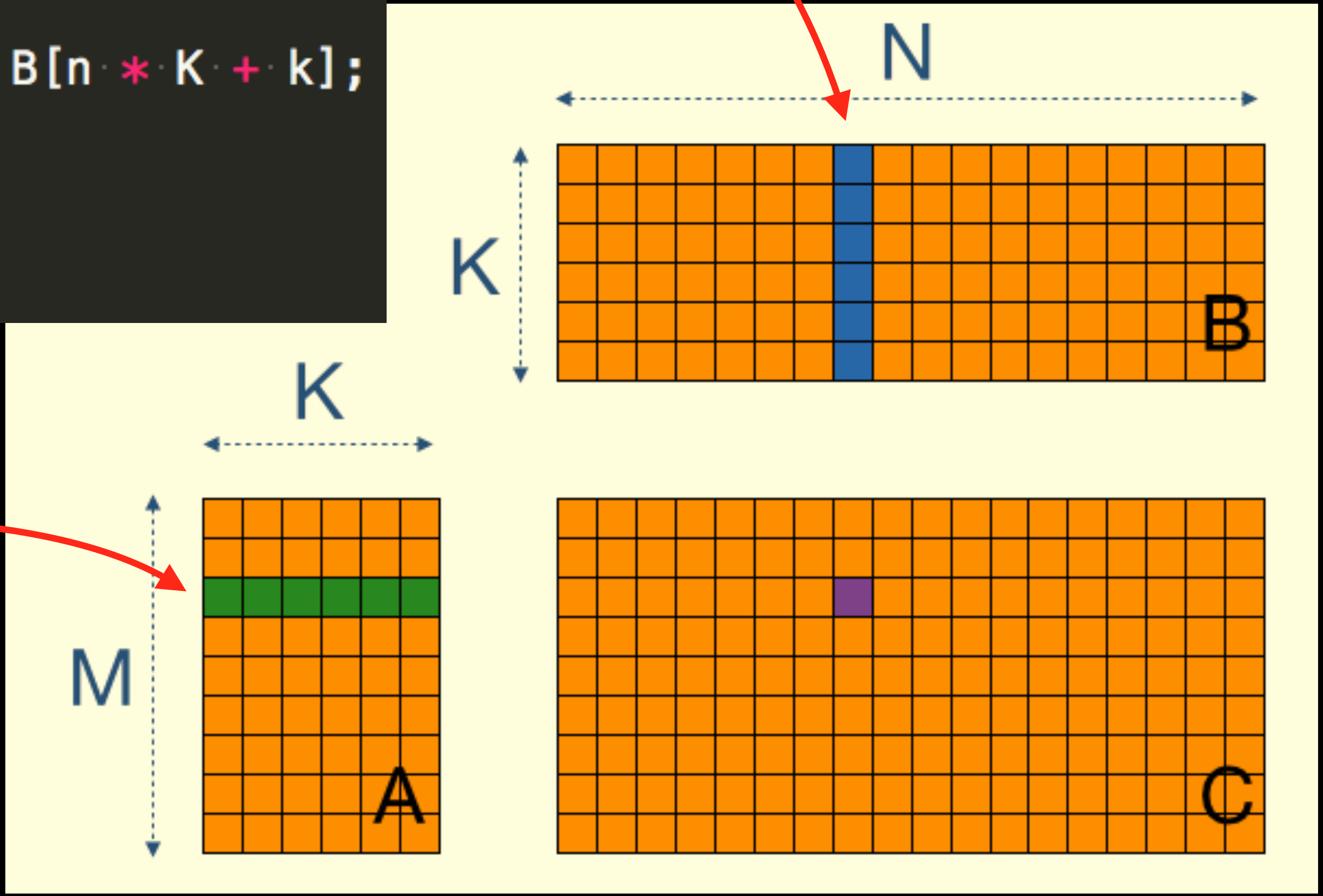
# Matrix-multiplication: C = A * B

www.cedricnugteren.nl/tutorial.php

```
for (int m = 0; m < M; ++m)
{
    for (int n = 0; n < N; ++n)
    {
        float acc = 0.0f;
        for (int k = 0; k < K; ++k)
        {
            acc += A[k * M + m] * B[n * K + k];
        }
        C[n * M + m] = acc;
    }
}
```

n = 7

m = 2



## Assumptions:

- Single-precision floating-point
- Layout: column-major ordering

# OpenCL SGEMM tuning

www.cedricnugteren.nl/tutorial.php

CUDA version is very similar

Optimisation steps:

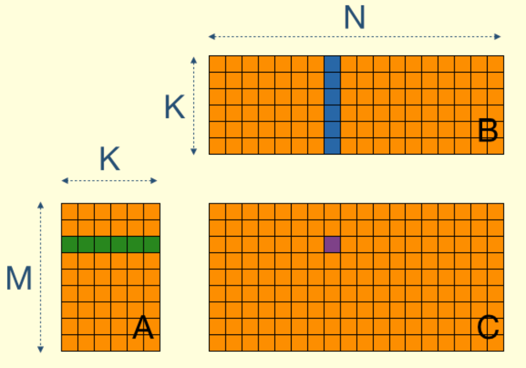
1. Naive implementation
2. Tiling in the shared memory
3. More work per thread
4. Wider data-types (vectors)
5. Transposed input matrix and rectangular tiles
6. 2D register blocking

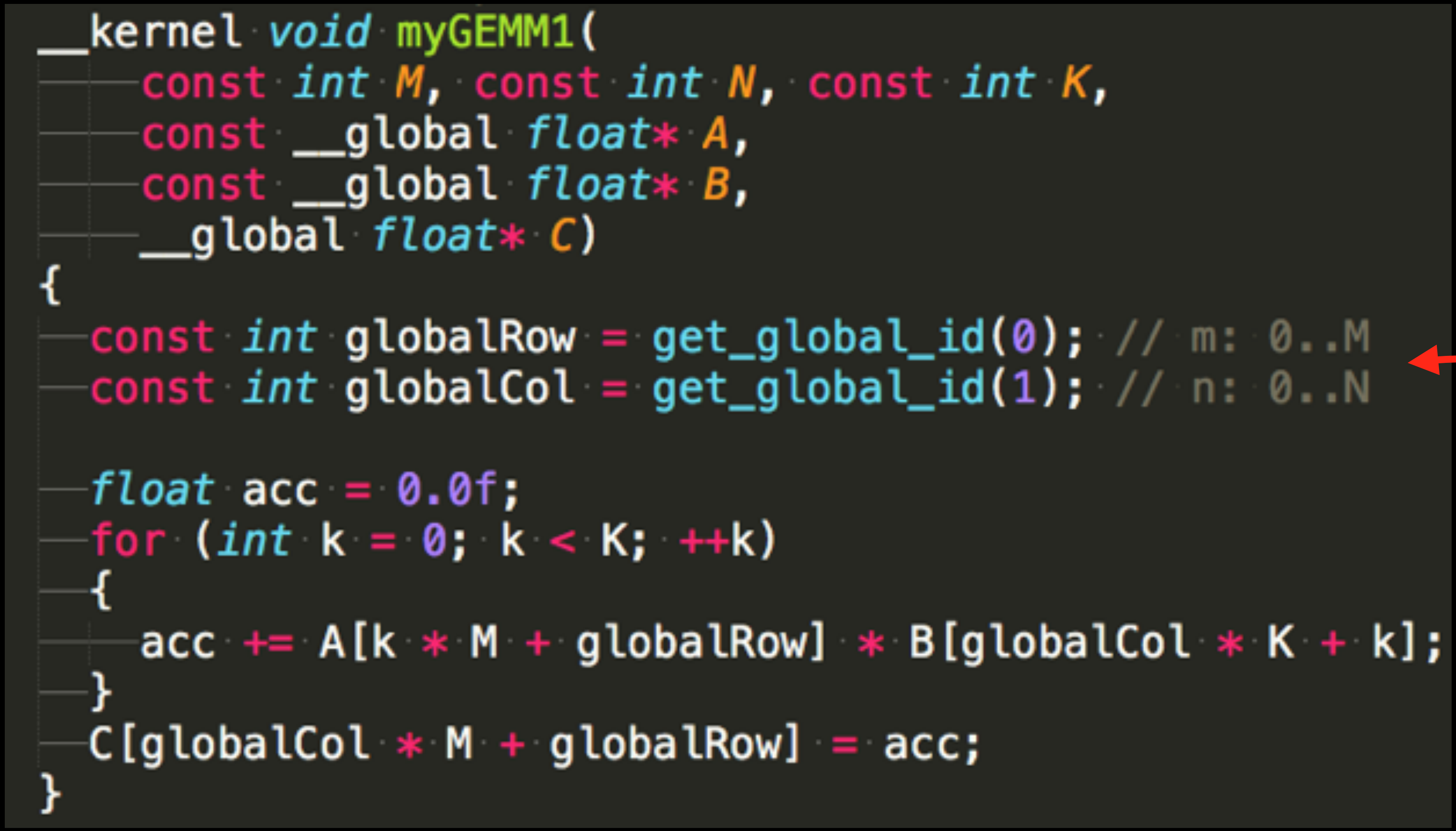
in short

Tesla K40m (Kepler), ECC on
CUDA 6.5

# Step 1: Naive implementation

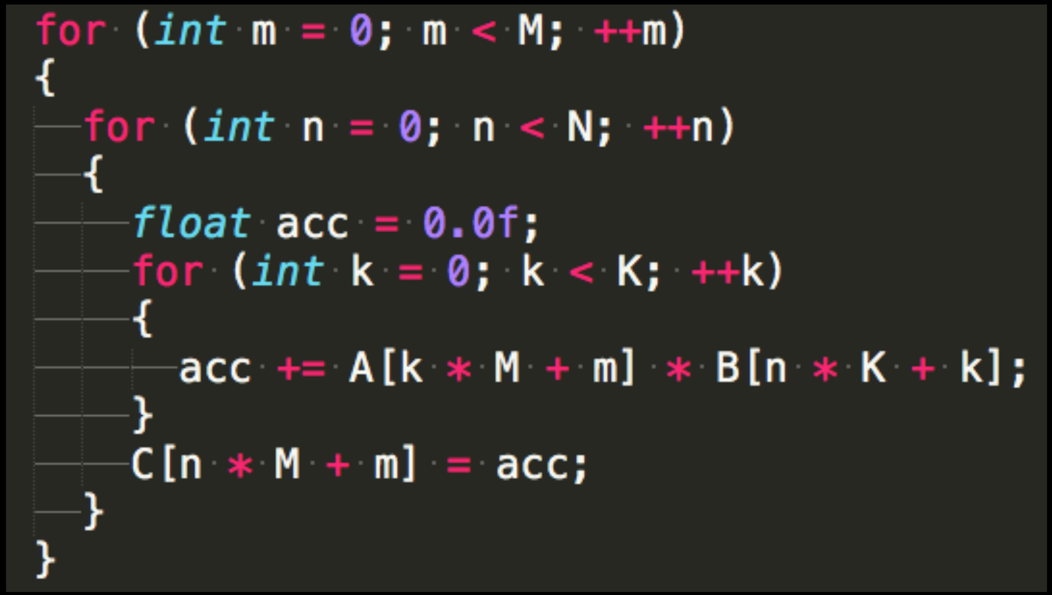
www.cedricnugteren.nl/tutorial.php





```
__kernel void myGEMM1(
        const int M, const int N, const int K,
        const __global float* A,
        const __global float* B,
        __global float* C)
{
    const int globalRow = get_global_id(0); // m: 0..M
    const int globalCol = get_global_id(1); // n: 0..N

    float acc = 0.0f;
    for (int k = 0; k < K; ++k)
    {
        acc += A[k * M + globalRow] * B[globalCol * K + k];
    }
    C[globalCol * M + globalRow] = acc;
}
```

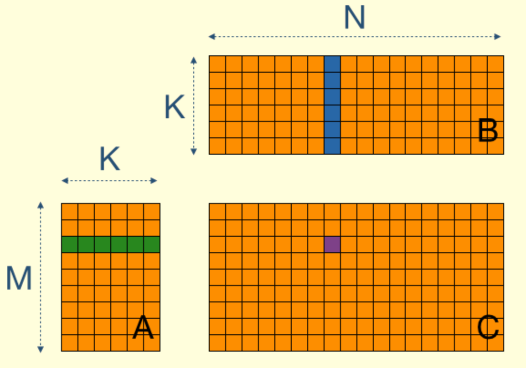
2D thread indexing

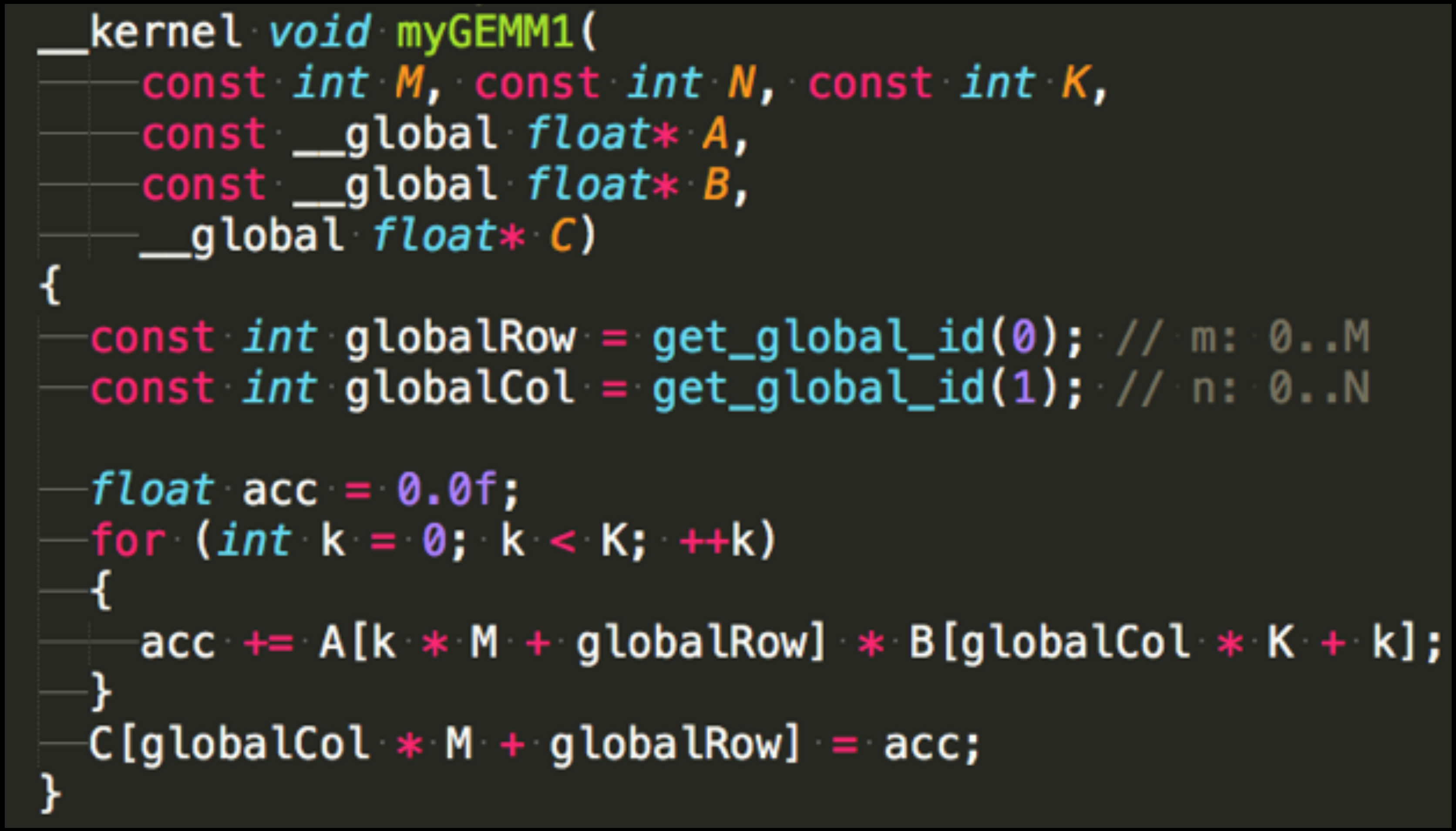


```
for (int m = 0; m < M; ++m)
{
    for (int n = 0; n < N; ++n)
    {
        float acc = 0.0f;
        for (int k = 0; k < K; ++k)
        {
            acc += A[k * M + m] * B[n * K + k];
        }
        C[n * M + m] = acc;
    }
}
```

# Step 1: Naive implementation

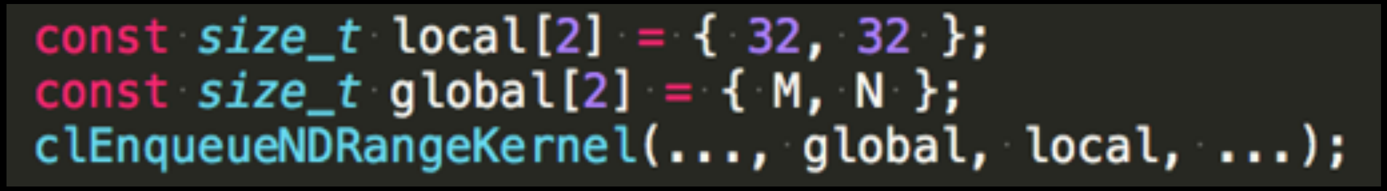
www.cedricnugteren.nl/tutorial.php



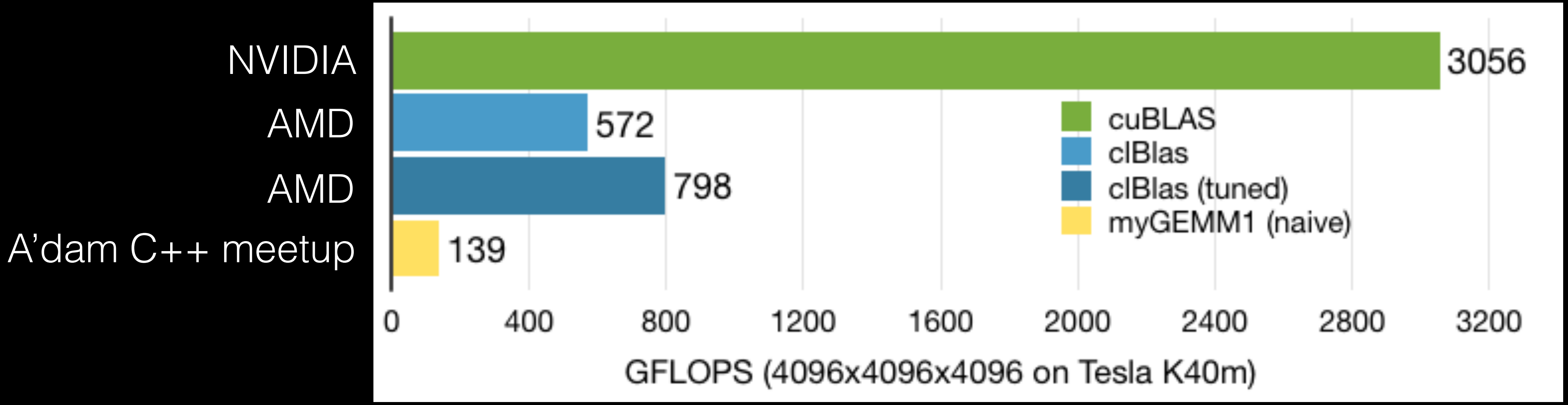


```
__kernel void myGEMM1(
        const int M, const int N, const int K,
        const __global float* A,
        const __global float* B,
        __global float* C)
{
    const int globalRow = get_global_id(0); // m: 0..M
    const int globalCol = get_global_id(1); // n: 0..N

    float acc = 0.0f;
    for (int k = 0; k < K; ++k)
    {
        acc += A[k * M + globalRow] * B[globalCol * K + k];
    }
    C[globalCol * M + globalRow] = acc;
}
```

1024 threads per block

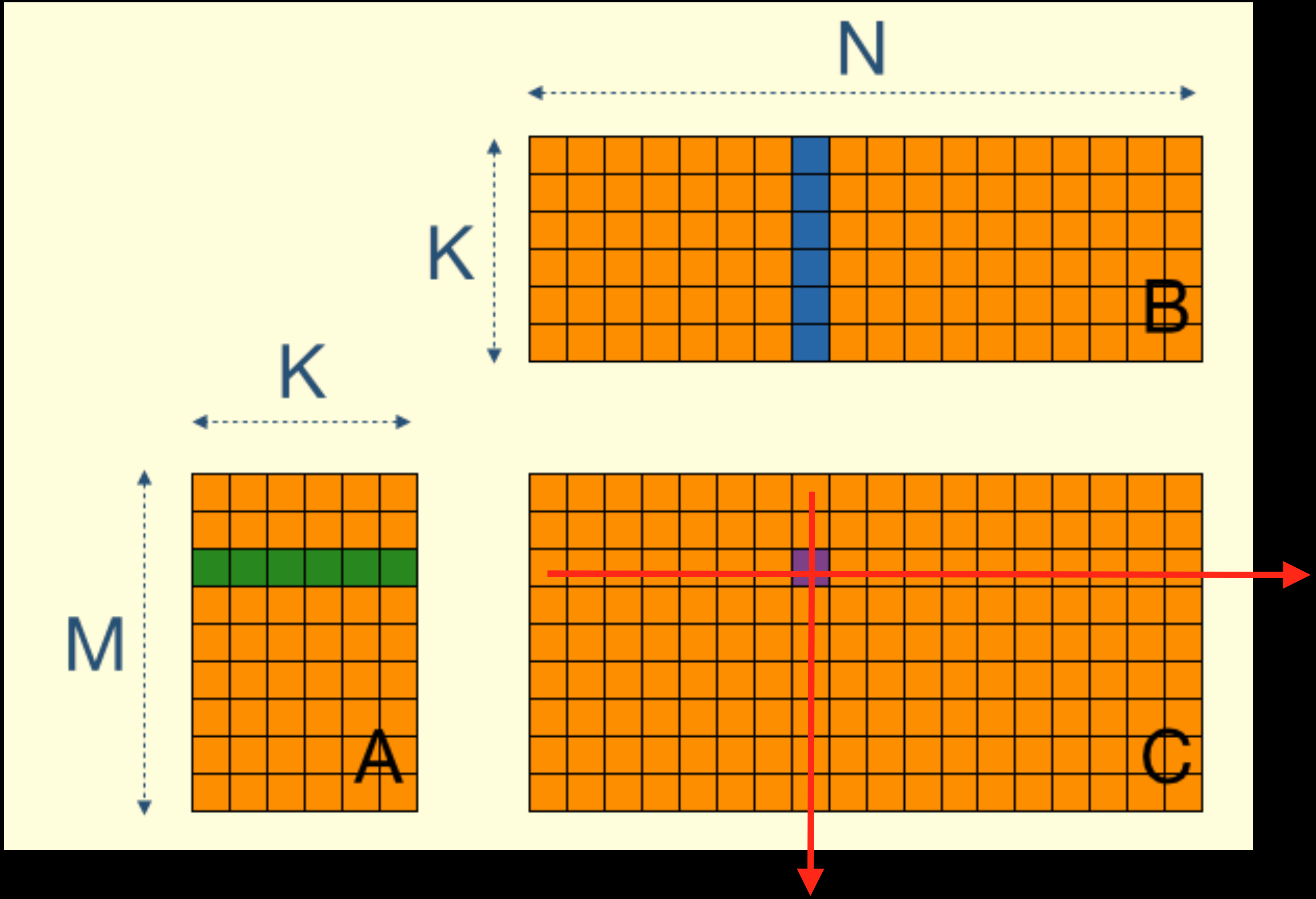


```
const size_t local[2] = { 32, 32 };
const size_t global[2] = { M, N };
clEnqueueNDRangeKernel(..., global, local, ...);
```



# Step 2: Tiling in the shared memory

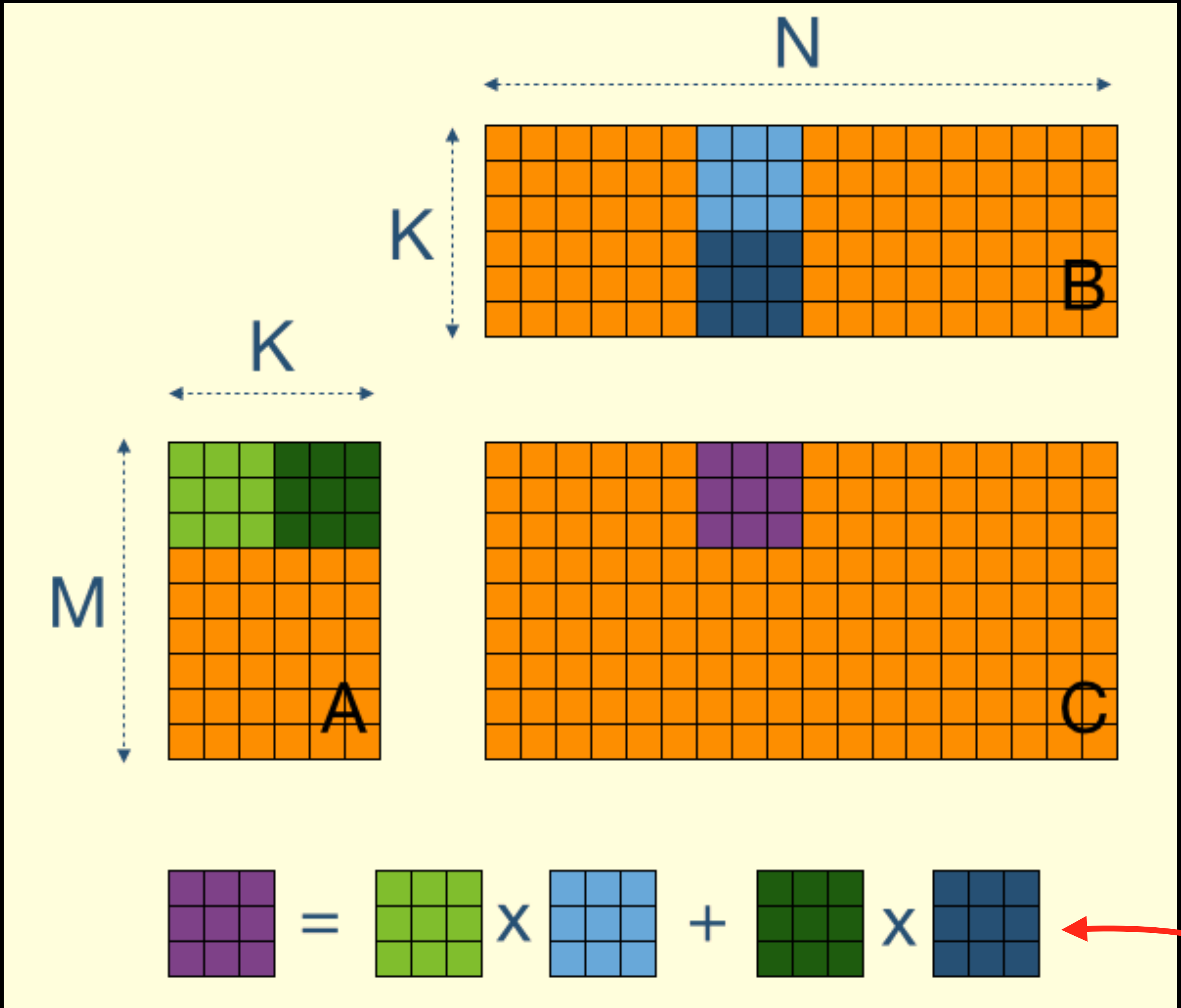
www.cedricnugteren.nl/tutorial.php



Lots of data re-use in A and B!

# Step 2: Tiling in the shared memory

www.cedricnugteren.nl/tutorial.php

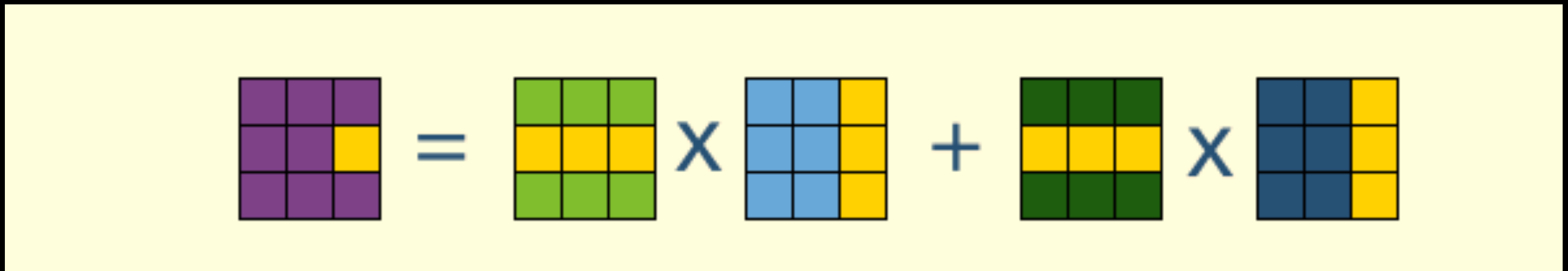


Shared memory required:
2 * 32 * 32 * 4B = 8KB

Shared memory available:
48K

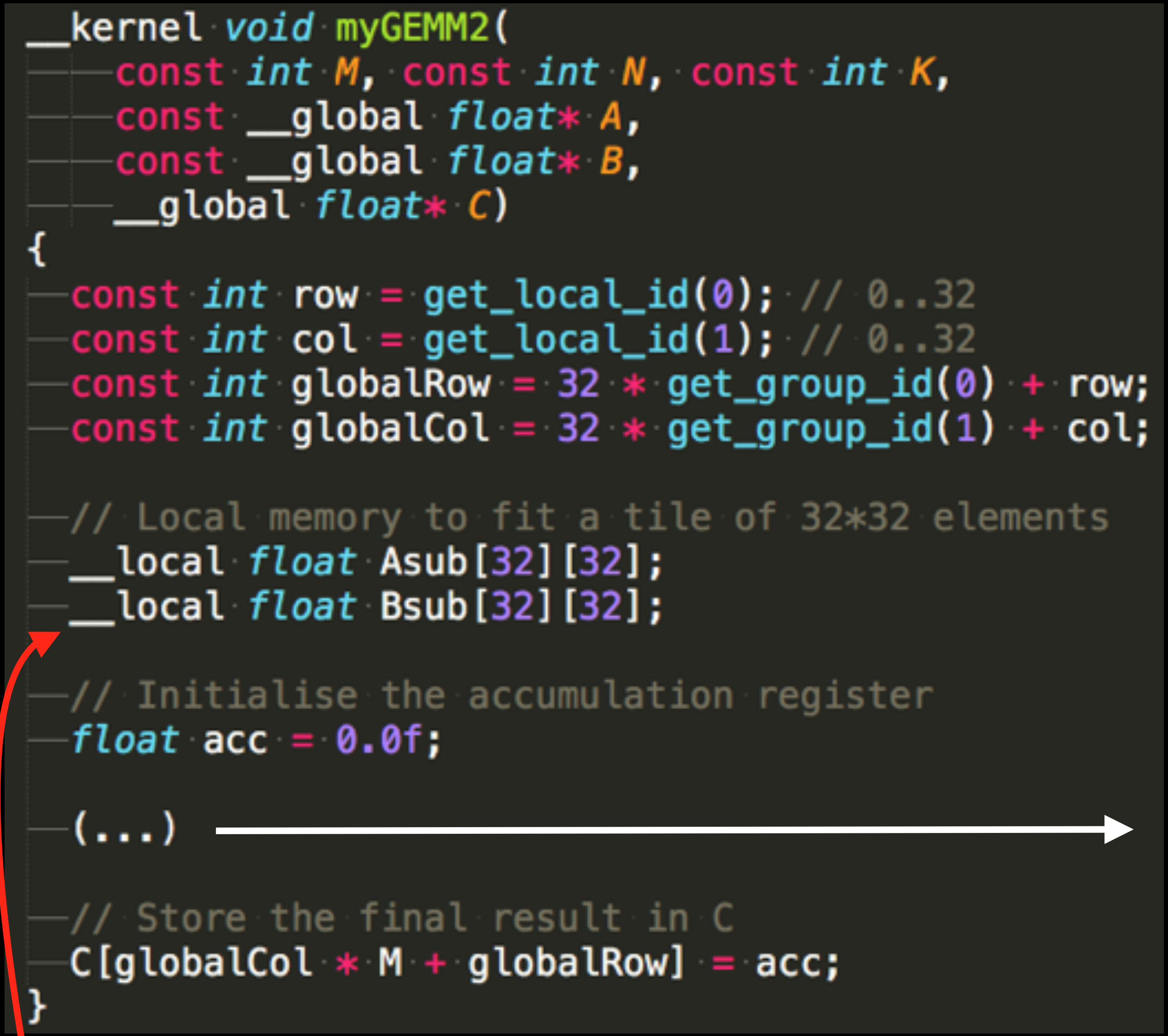
Active blocks per core/SM:
48/8 = 6

32 by 32



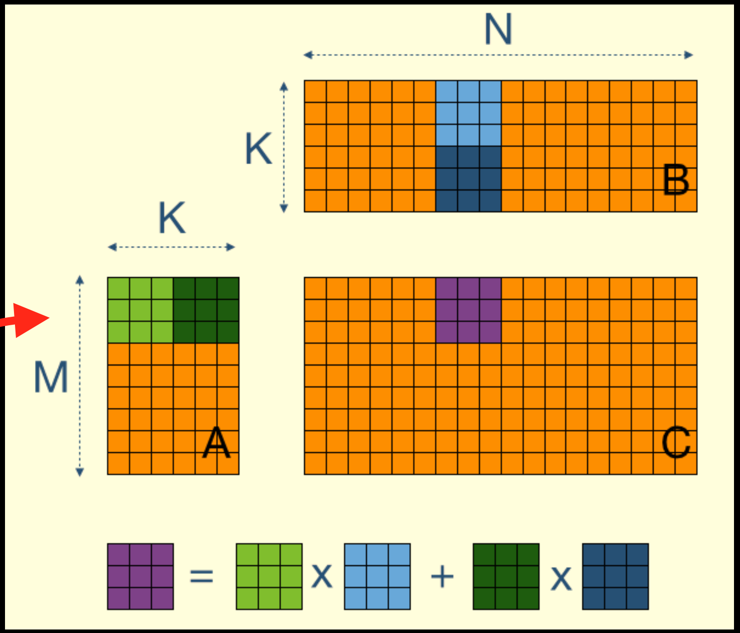
# Step 2: Tiling in the shared memory

www.cedricnugteren.nl/tutorial.php

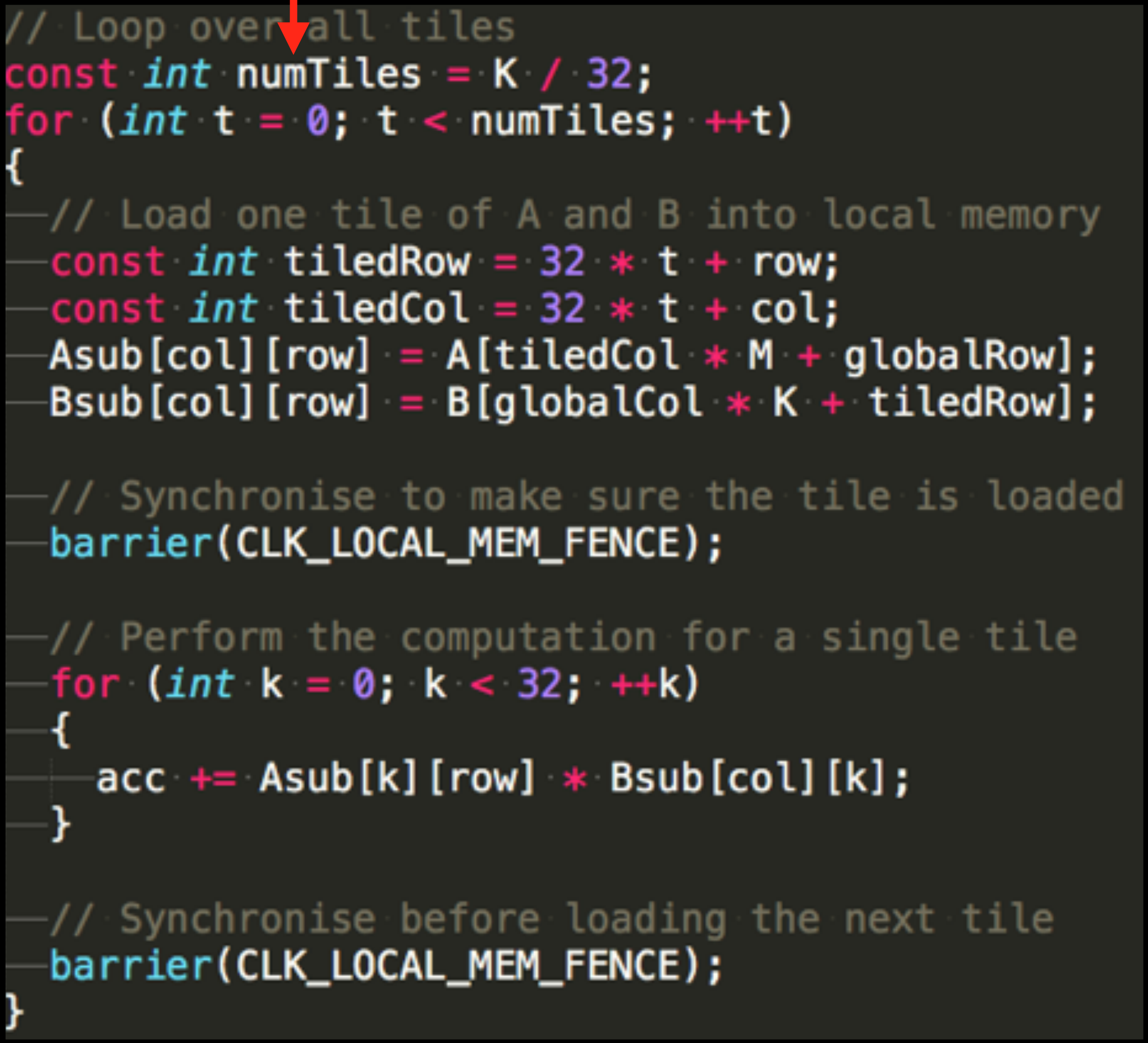


```
__kernel void myGEMM2(
    const int M, const int N, const int K,
    const __global float* A,
    const __global float* B,
    __global float* C)
{
  const int row = get_local_id(0); // 0..32
  const int col = get_local_id(1); // 0..32
  const int globalRow = 32 * get_group_id(0) + row;
  const int globalCol = 32 * get_group_id(1) + col;

  // Local memory to fit a tile of 32*32 elements
  __local float Asub[32][32];
  __local float Bsub[32][32];

  // Initialise the accumulation register
  float acc = 0.0f;

  (...)

  // Store the final result in C
  C[globalCol * M + globalRow] = acc;
}
```

Shared within a
thread-block



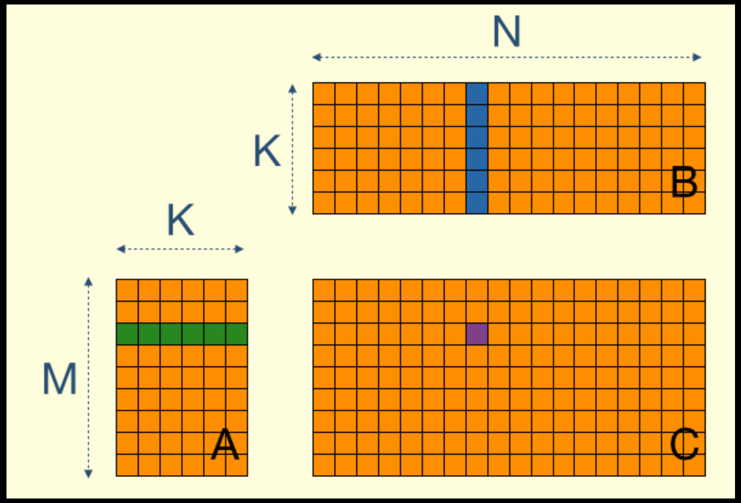
2 in example

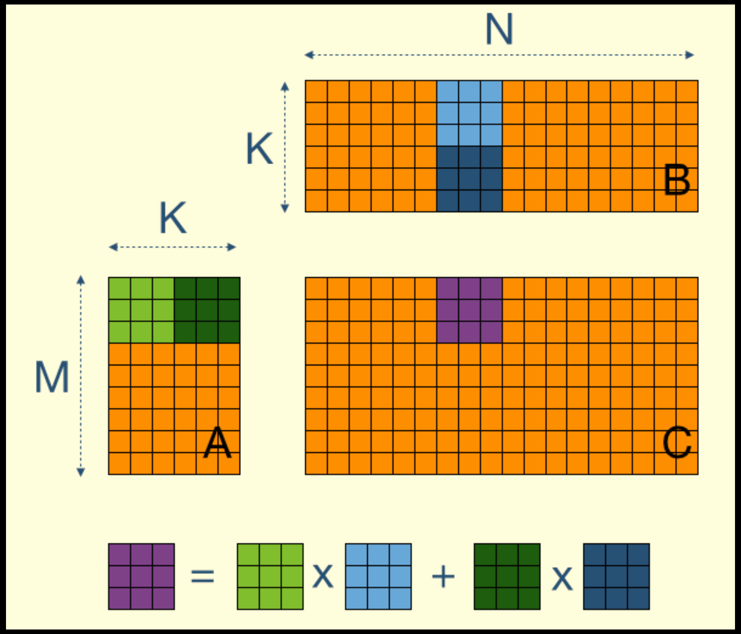


```
// Loop over all tiles
const int numTiles = K / 32;
for (int t = 0; t < numTiles; ++t)
{
  // Load one tile of A and B into local memory
  const int tiledRow = 32 * t + row;
  const int tiledCol = 32 * t + col;
  Asub[col][row] = A[tiledCol * M + globalRow];
  Bsub[col][row] = B[globalCol * K + tiledRow];

  // Synchronise to make sure the tile is loaded
  barrier(CLK_LOCAL_MEM_FENCE);

  // Perform the computation for a single tile
  for (int k = 0; k < 32; ++k)
  {
    acc += Asub[k][row] * Bsub[col][k];
  }

  // Synchronise before loading the next tile
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

# Step 2: Tiling in the local memory

www.cedricnugteren.nl/tutorial.php





Factor 3 in picture:

- Before 2x6 loads
- Now 2x2 loads

Advantages:

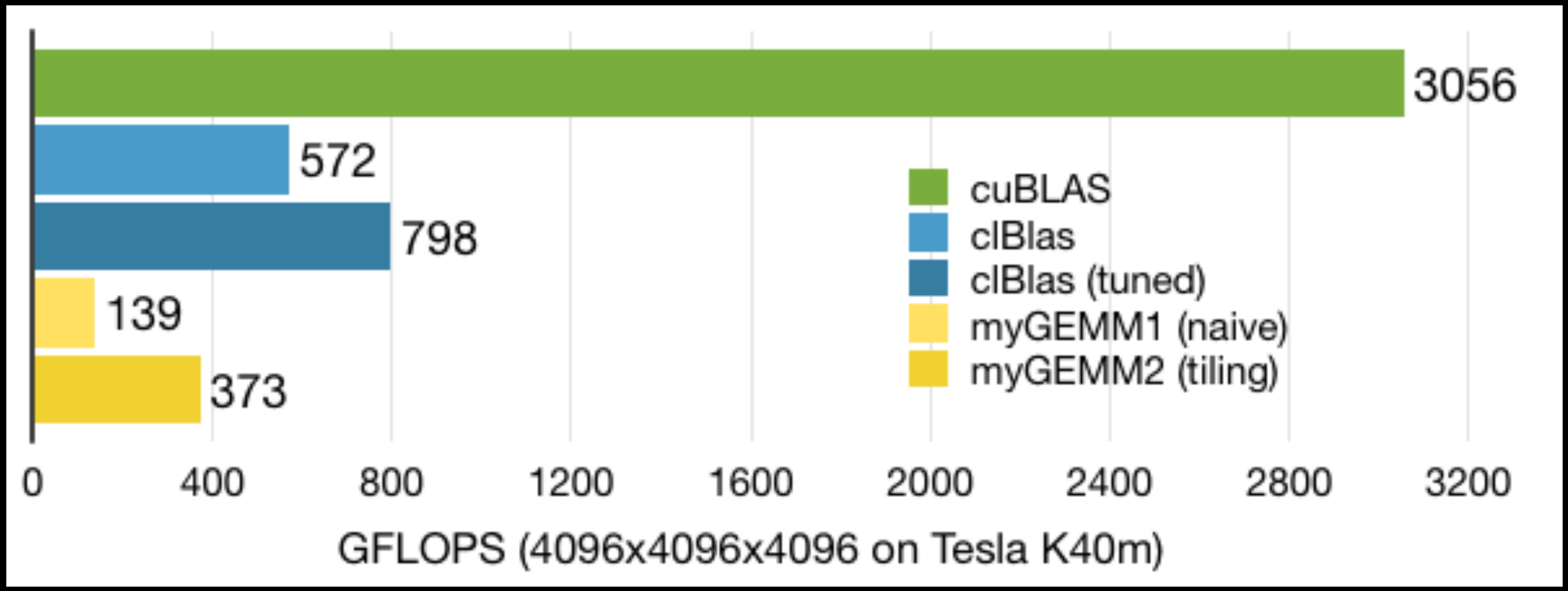
1. Reduction of 32x in off-chip memory accesses
2. Coalesced memory accesses now also for B

NVIDIA

AMD

AMD

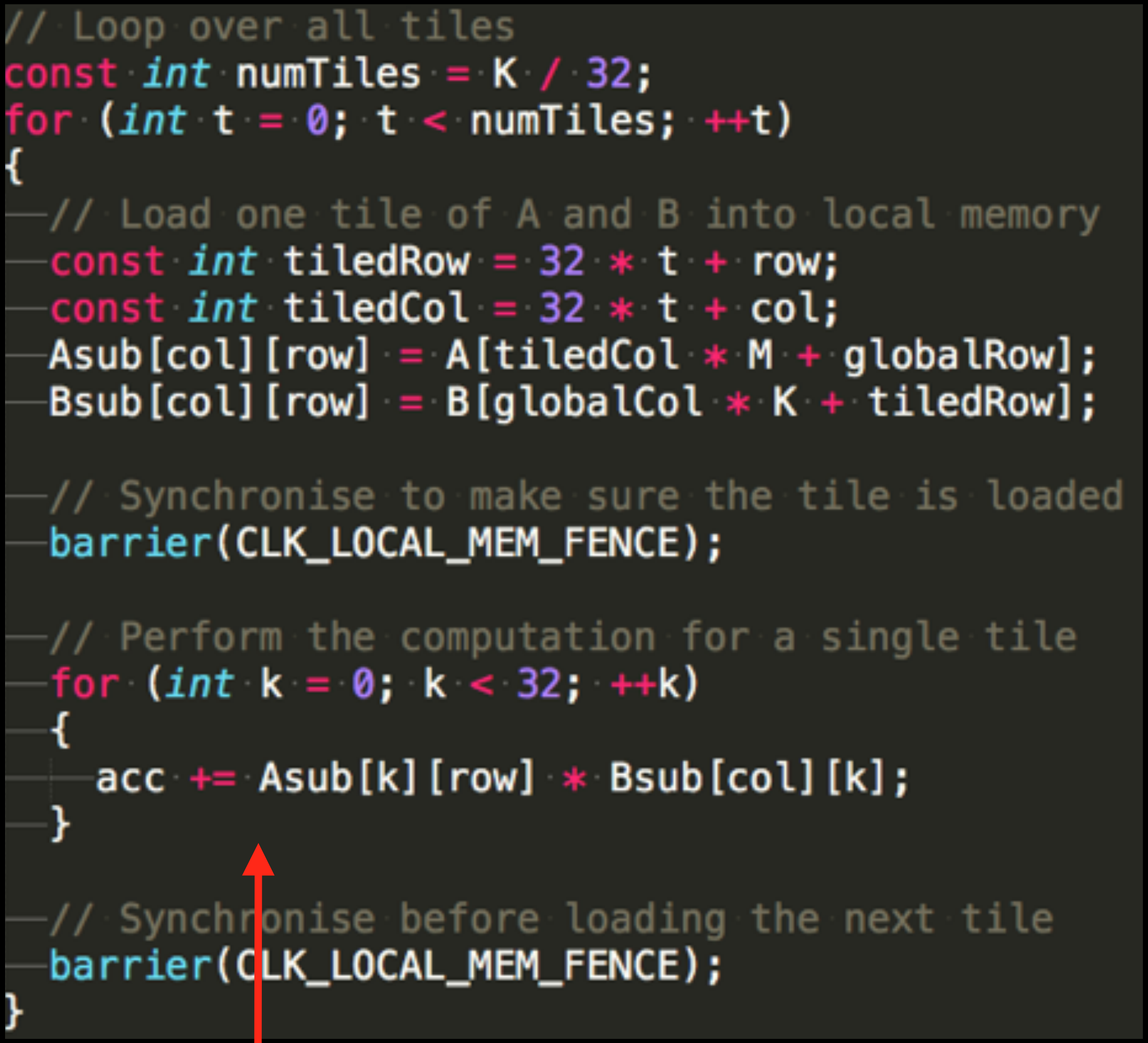
A'dam C++ meetup



# Step 3: More work per thread

www.cedricnugteren.nl/tutorial.php

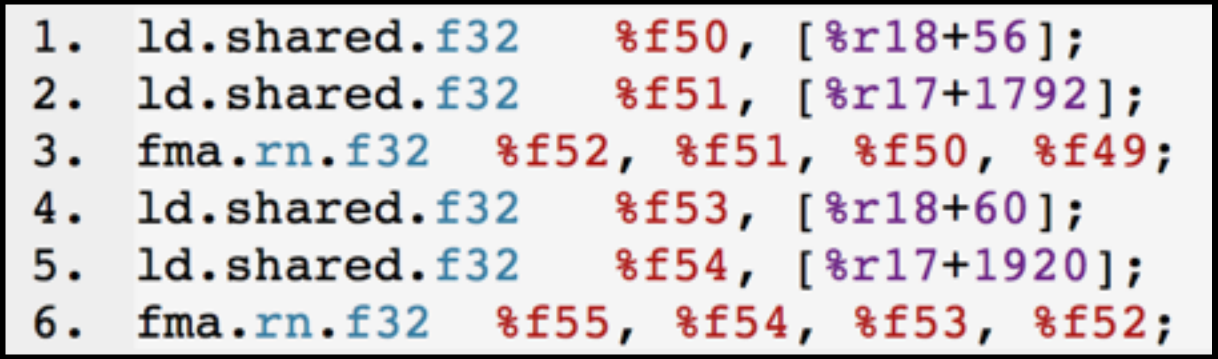
Main body of our kernel:



```
// Loop over all tiles
const int numTiles = K / 32;
for (int t = 0; t < numTiles; ++t)
{
    // Load one tile of A and B into local memory
    const int tiledRow = 32 * t + row;
    const int tiledCol = 32 * t + col;
    Asub[col][row] = A[tiledCol * M + globalRow];
    Bsub[col][row] = B[globalCol * K + tiledRow];

    // Synchronise to make sure the tile is loaded
    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the computation for a single tile
    for (int k = 0; k < 32; ++k)
    {
        acc += Asub[k][row] * Bsub[col][k];
    }

    // Synchronise before loading the next tile
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Unroll this loop

Two loop iterations:



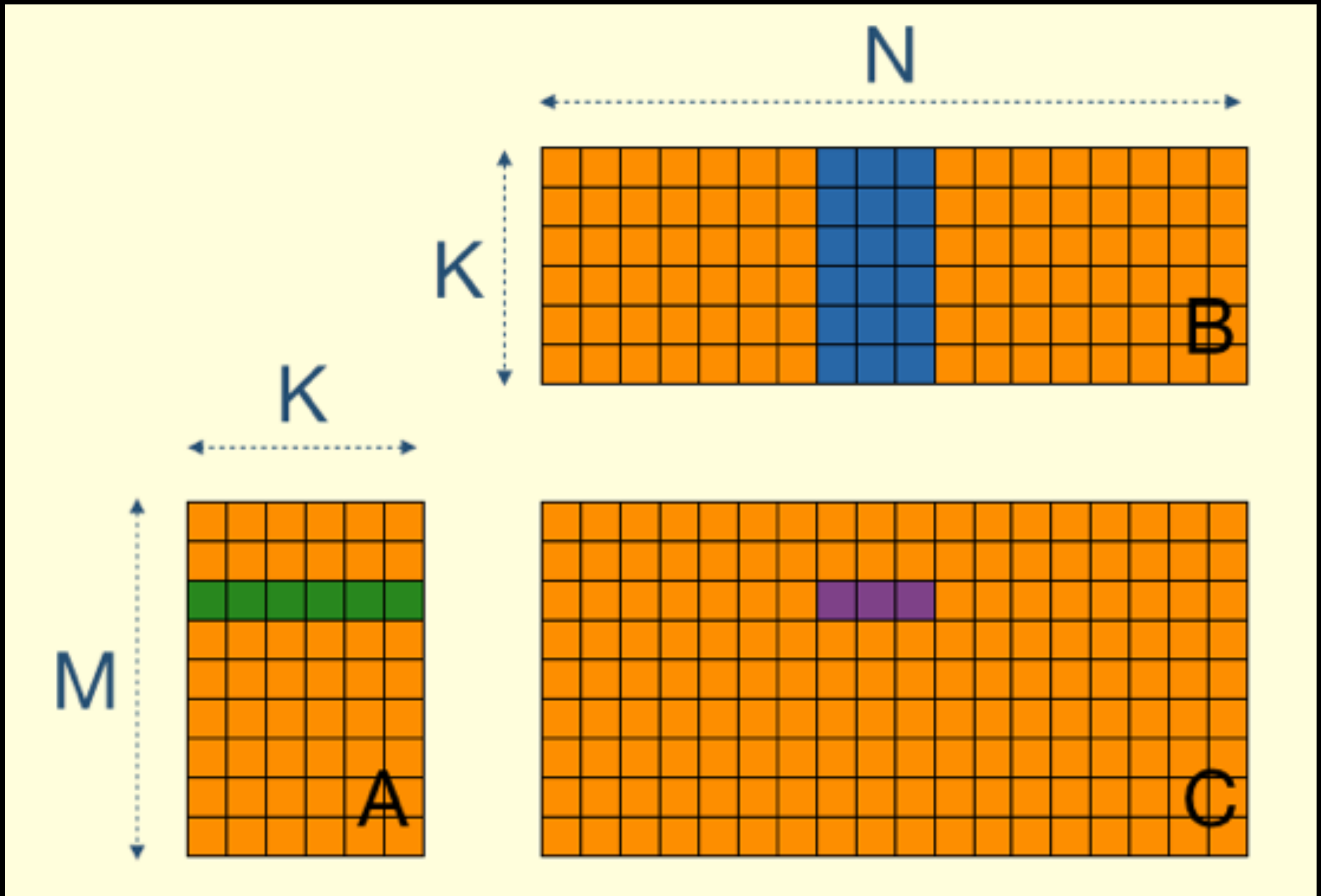
```
1.  ld.shared.f32    %f50, [%r18+56];
2.  ld.shared.f32    %f51, [%r17+1792];
3.  fma.rn.f32  %f52, %f51, %f50, %f49;
4.  ld.shared.f32    %f53, [%r18+60];
5.  ld.shared.f32    %f54, [%r17+1920];
6.  fma.rn.f32  %f55, %f54, %f53, %f52;
```

1. Load `Asub[k][row]` into a register
2. Load `Bsub[col][k]` into a register
3. Perform fused multiply-add (FMA)
4. Load `Asub[k+1][row]` into a register
5. Load `Bsub[col][k+1]` into a register
6. Perform fused multiply-add (FMA)

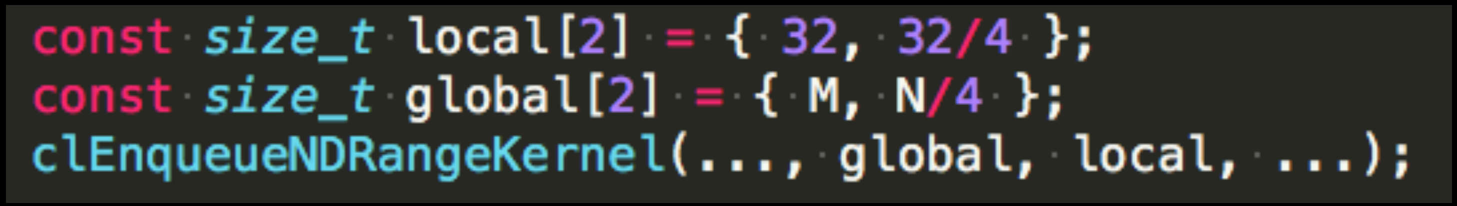
Not going to get peak GFLOPS :-(

# Step 3: More work per thread

www.cedricnugteren.nl/tutorial.php



Similar idea as before, but now to save on-chip memory accesses

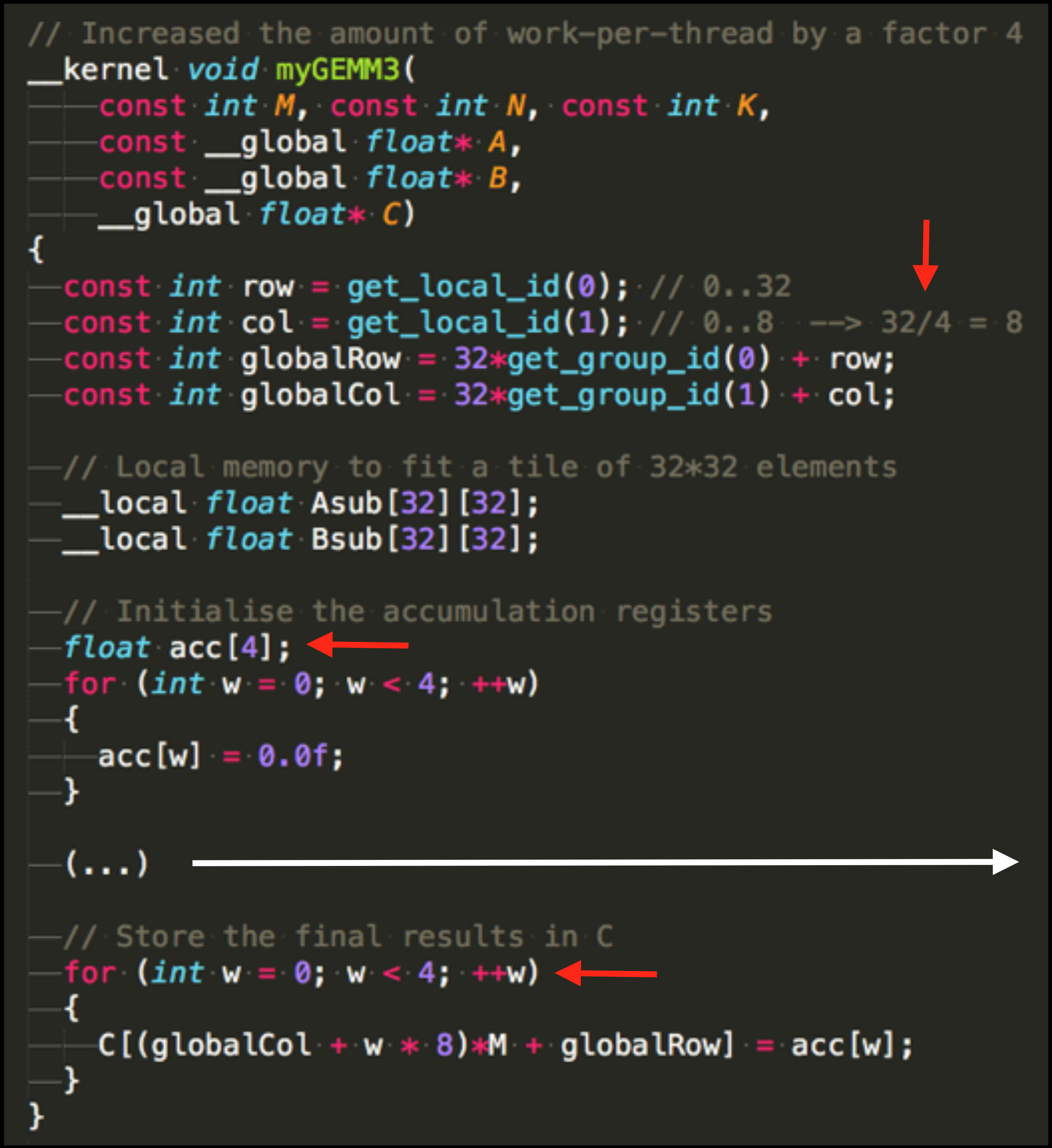


```
const size_t local[2] = { 32, 32/4 };
const size_t global[2] = { M, N/4 };
clEnqueueNDRangeKernel(..., global, local, ...);
```

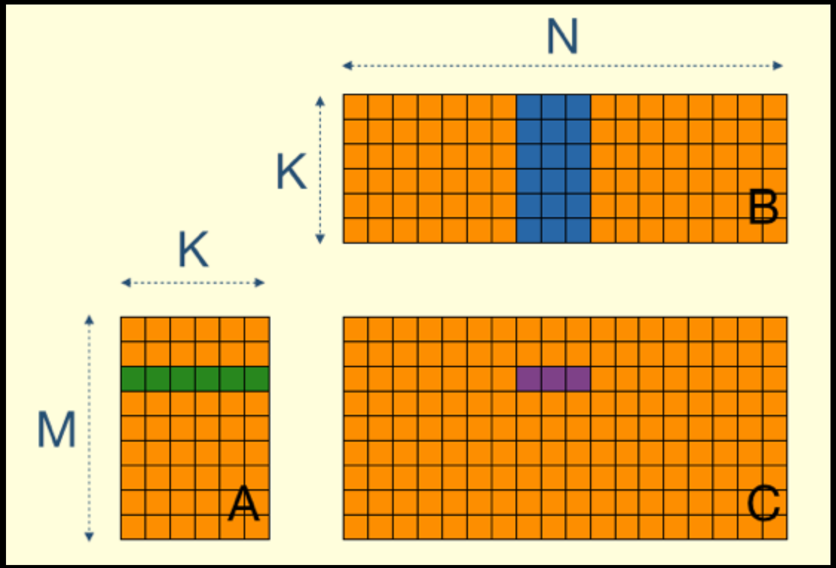
Tile: 32 by 32
Work-per-thread: 4

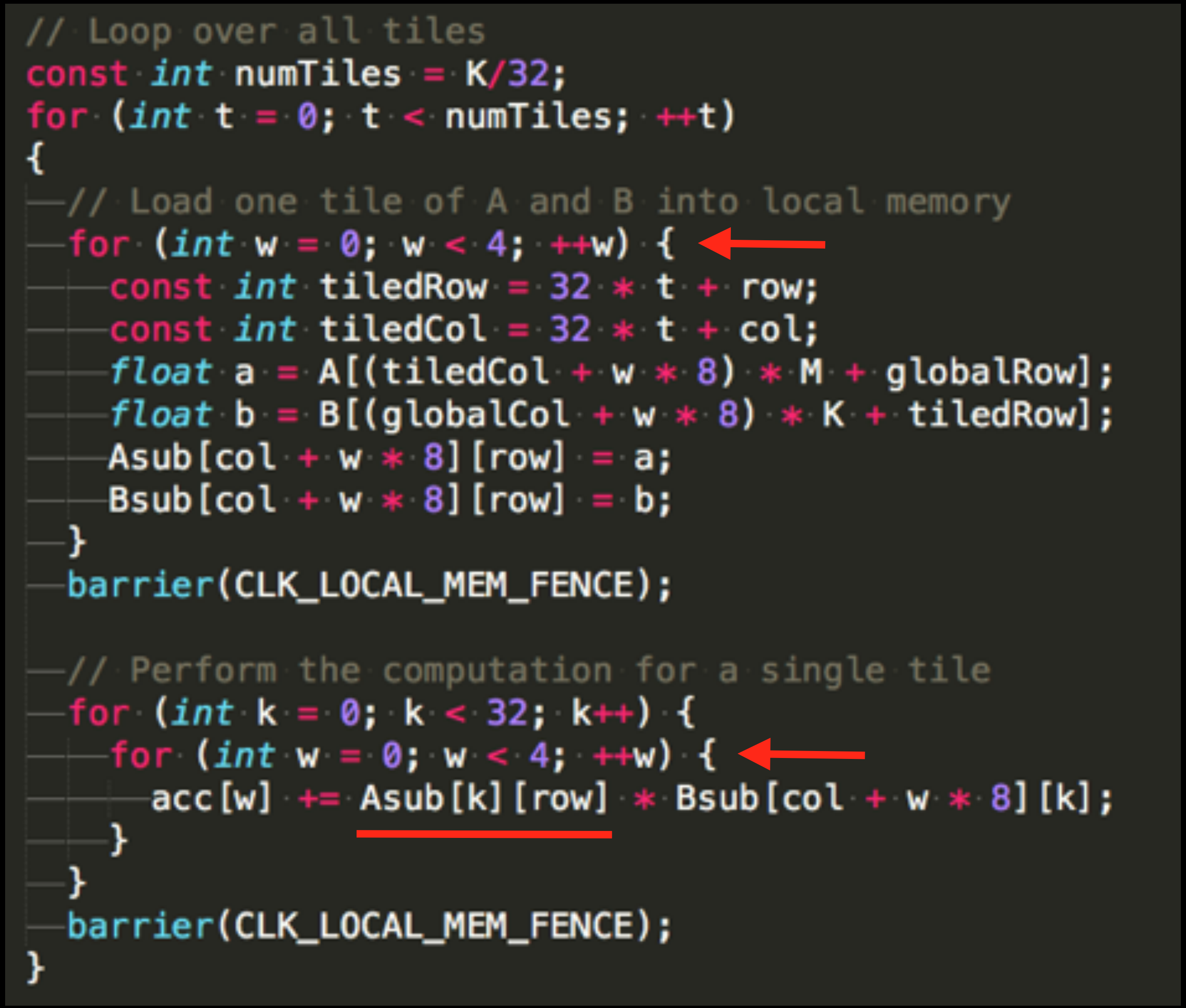
# Step 3: More work per thread

www.cedricnugteren.nl/tutorial.php



```
// Increased the amount of work-per-thread by a factor 4
__kernel void myGEMM3(
        const int M, const int N, const int K,
        const __global float* A,
        const __global float* B,
        __global float* C)
{
    const int row = get_local_id(0); // 0..32
    const int col = get_local_id(1); // 0..8  --> 32/4 = 8
    const int globalRow = 32*get_group_id(0) + row;
    const int globalCol = 32*get_group_id(1) + col;

    // Local memory to fit a tile of 32*32 elements
    __local float Asub[32][32];
    __local float Bsub[32][32];

    // Initialise the accumulation registers
    float acc[4];
    for (int w = 0; w < 4; ++w)
    {
        acc[w] = 0.0f;
    }

    (...)

    // Store the final results in C
    for (int w = 0; w < 4; ++w)
    {
        C[(globalCol + w * 8)*M + globalRow] = acc[w];
    }
}
```



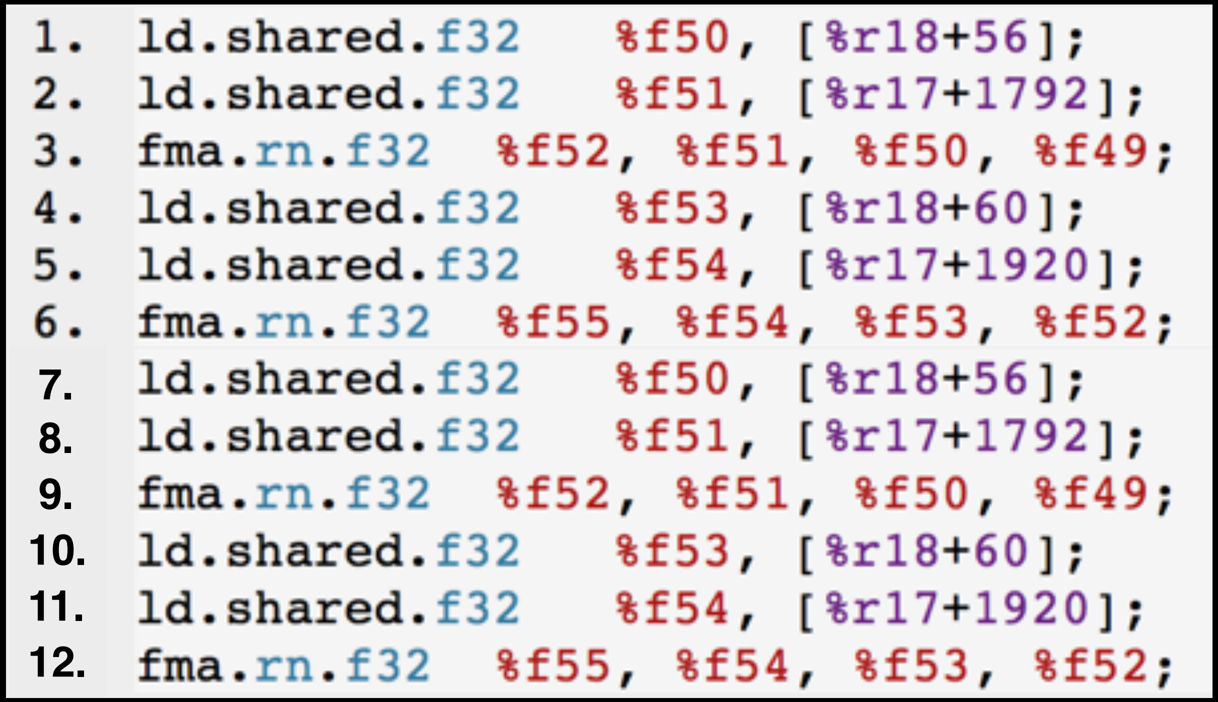


```
// Loop over all tiles
const int numTiles = K/32;
for (int t = 0; t < numTiles; ++t)
{
    // Load one tile of A and B into local memory
    for (int w = 0; w < 4; ++w) {
        const int tiledRow = 32 * t + row;
        const int tiledCol = 32 * t + col;
        float a = A[(tiledCol + w * 8) * M + globalRow];
        float b = B[(globalCol + w * 8) * K + tiledRow];
        Asub[col + w * 8][row] = a;
        Bsub[col + w * 8][row] = b;
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the computation for a single tile
    for (int k = 0; k < 32; k++) {
        for (int w = 0; w < 4; ++w) {
            acc[w] += Asub[k][row] * Bsub[col + w * 8][k];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

# Step 3: More work per thread

www.cedricnugteren.nl/tutorial.php

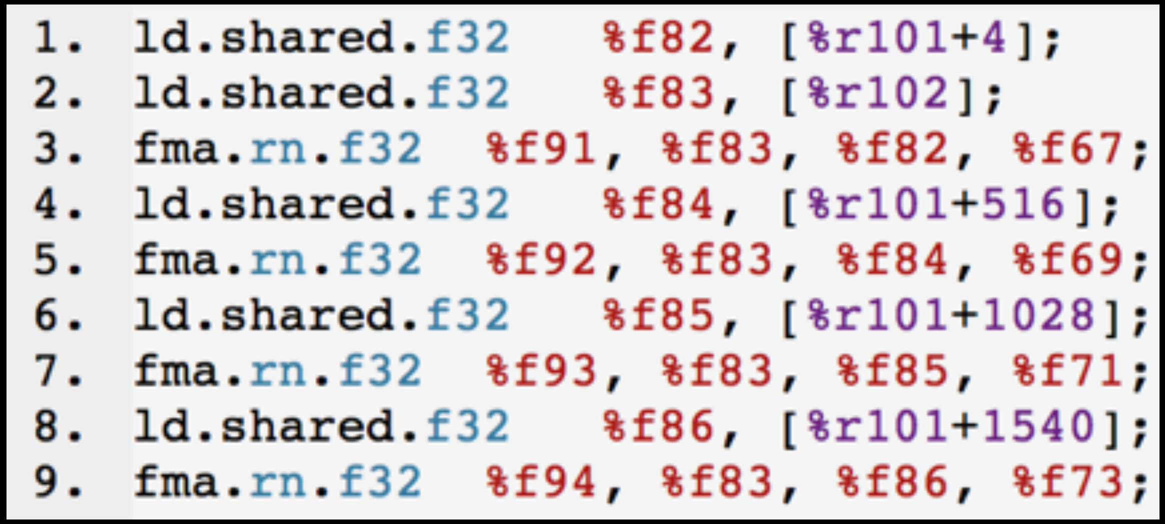
Previous version (4 iterations):



```
ld.shared.f32   %f50, [%r18+56];
ld.shared.f32   %f51, [%r17+1792];
fma.rn.f32  %f52, %f51, %f50, %f49;
ld.shared.f32   %f53, [%r18+60];
ld.shared.f32   %f54, [%r17+1920];
fma.rn.f32  %f55, %f54, %f53, %f52;
ld.shared.f32   %f50, [%r18+56];
ld.shared.f32   %f51, [%r17+1792];
fma.rn.f32  %f52, %f51, %f50, %f49;
ld.shared.f32   %f53, [%r18+60];
ld.shared.f32   %f54, [%r17+1920];
fma.rn.f32  %f55, %f54, %f53, %f52;
```

8 loads for 4 FMA

New version (4 iterations)

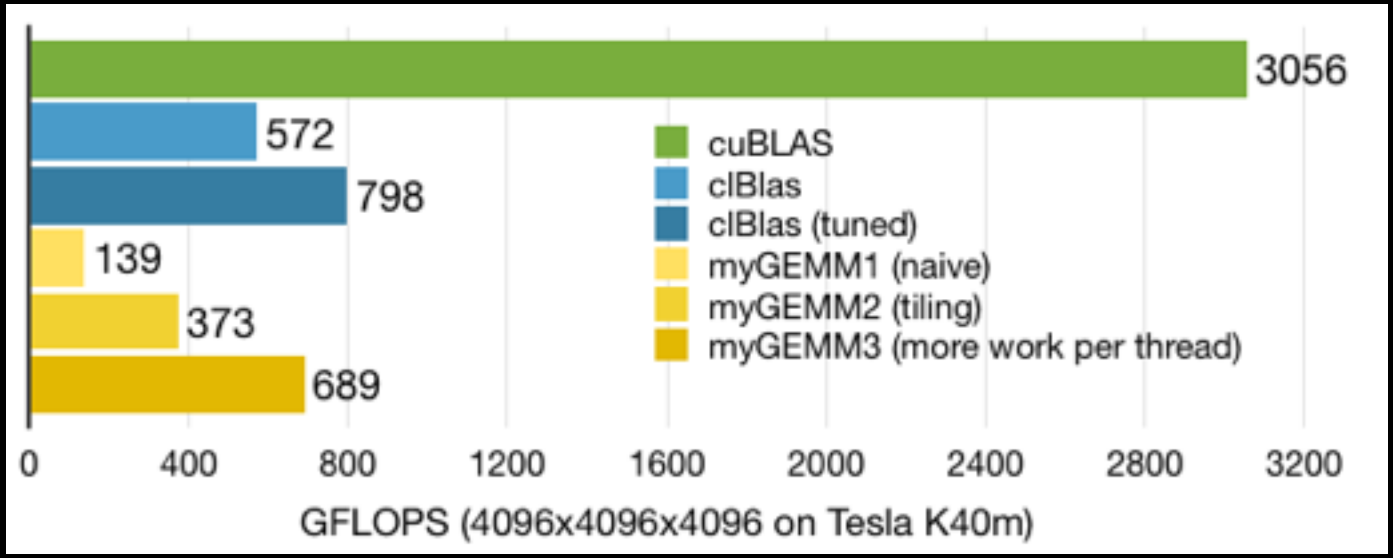


```
ld.shared.f32   %f82, [%r101+4];
ld.shared.f32   %f83, [%r102];
fma.rn.f32  %f91, %f83, %f82, %f67;
ld.shared.f32   %f84, [%r101+516];
fma.rn.f32  %f92, %f83, %f84, %f69;
ld.shared.f32   %f85, [%r101+1028];
fma.rn.f32  %f93, %f83, %f85, %f71;
ld.shared.f32   %f86, [%r101+1540];
fma.rn.f32  %f94, %f83, %f86, %f73;
```

(4+1) loads for 4 FMA

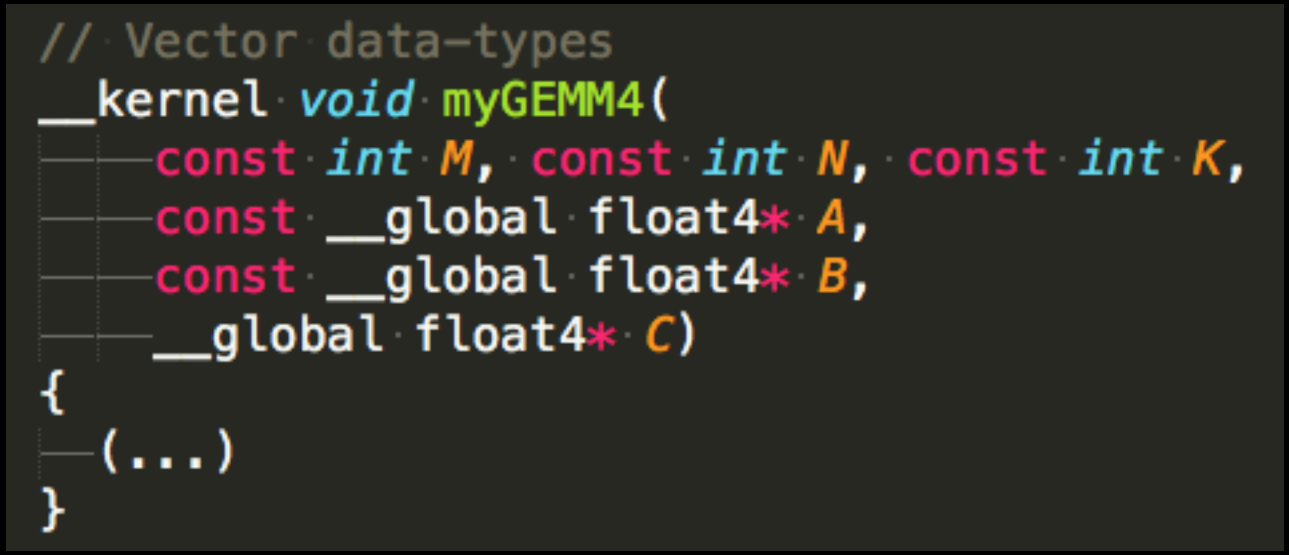
NVIDIA
AMD
AMD

A'dam C++ meetup



# Step 4: Wider data-types

www.cedricnugteren.nl/tutorial.php

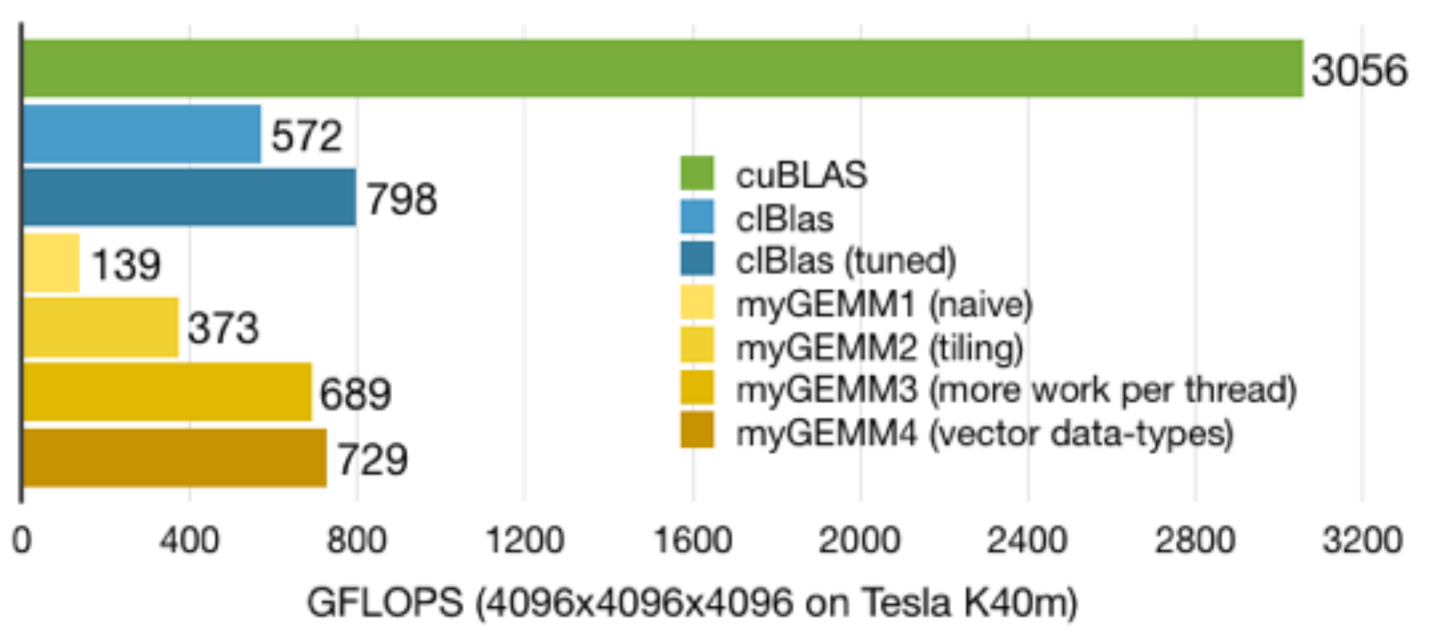


```
// Vector data-types
__kernel void myGEMM4(
        const int M, const int N, const int K,
        const __global float4* A,
        const __global float4* B,
        __global float4* C)
{
    (...)
}
```

Vector operations and loads/stores:

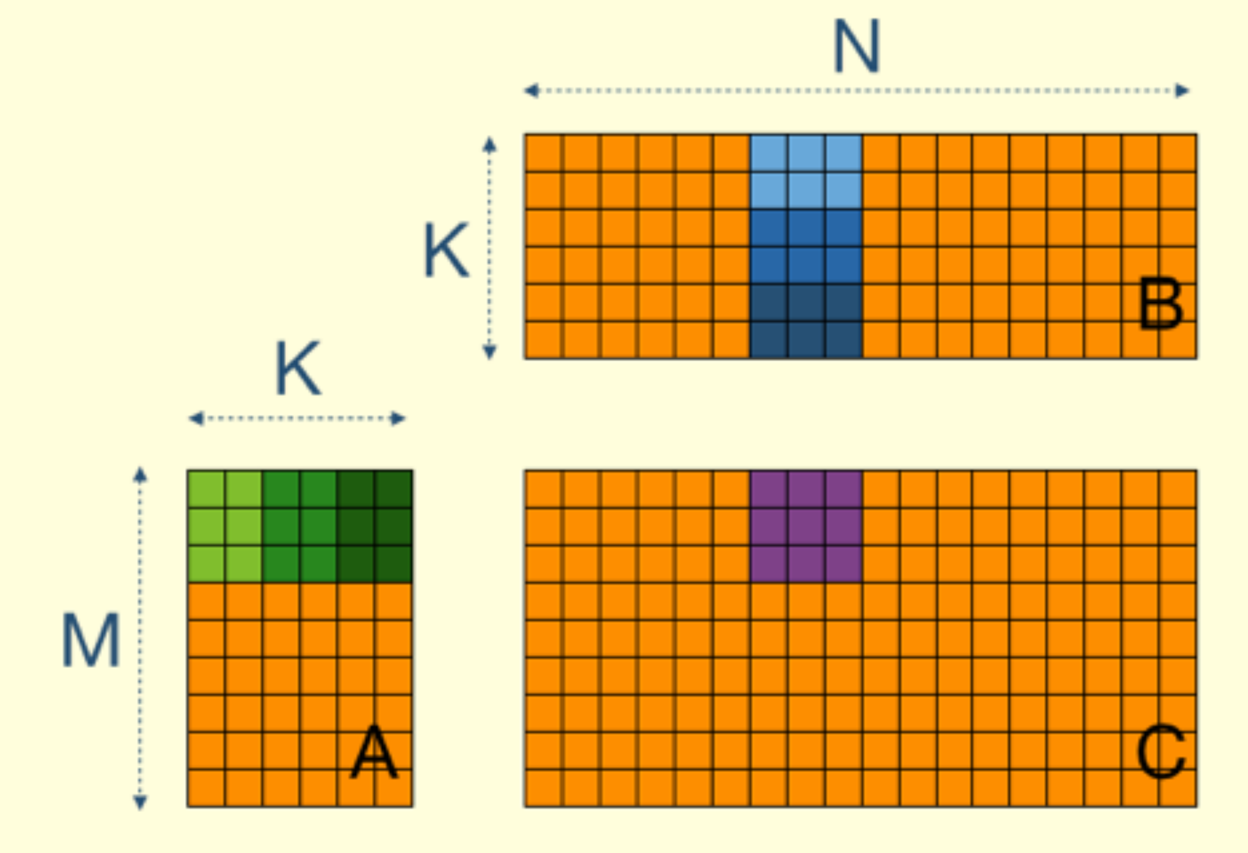
1. Not so useful for:
    1. NVIDIA GPUs
    2. Modern AMD GPUs

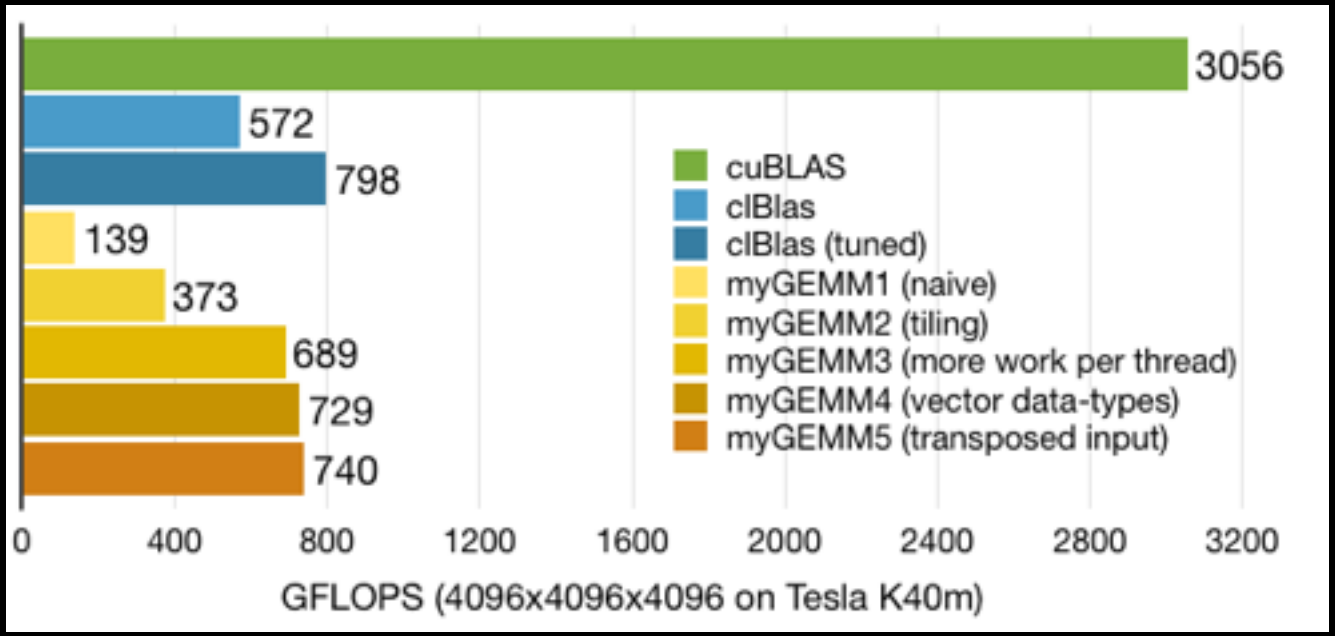
2. Important for:
    1. Older AMD GPUs (VLIW)
    2. Intel Xeon Phi
    3. CPUs (NEON / SSE / AVX)



# Step 5: Pre-transpose input matrix

www.cedricnugteren.nl/tutorial.php





pre-transpose matrix B

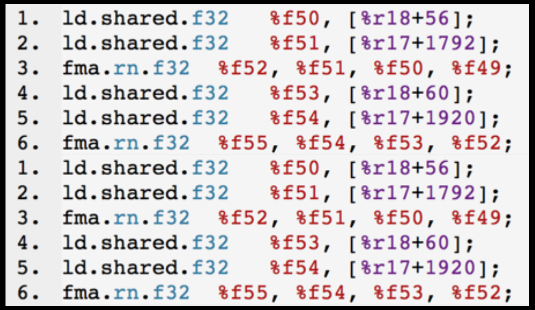
rectangular tiles
are now allowed

more tuning
opportunities

# Step 6: 2D register blocking

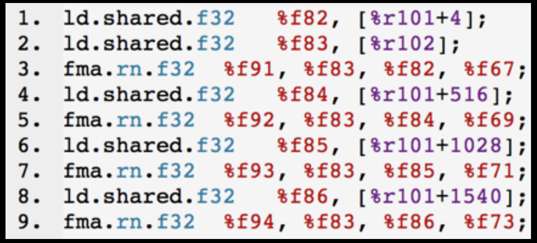
www.cedricnugteren.nl/tutorial.php

8 loads for 4 FMA

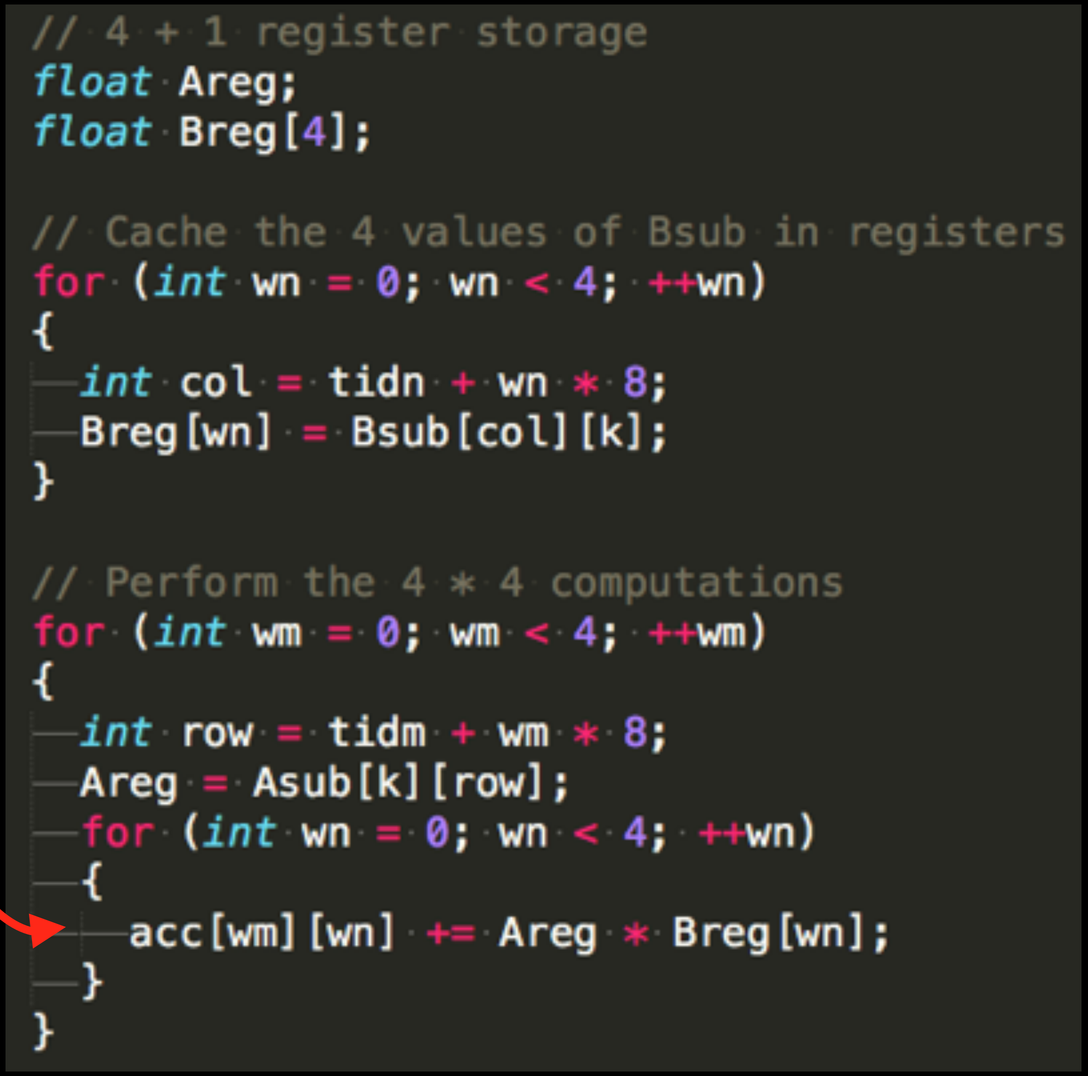


```
ld.shared.f32   %f50, [%r18+56];
ld.shared.f32   %f51, [%r17+1792];
fma.rn.f32  %f52, %f51, %f50, %f49;
ld.shared.f32   %f53, [%r18+60];
ld.shared.f32   %f54, [%r17+1920];
fma.rn.f32  %f55, %f54, %f53, %f52;
ld.shared.f32   %f50, [%r18+56];
ld.shared.f32   %f51, [%r17+1792];
fma.rn.f32  %f52, %f51, %f50, %f49;
ld.shared.f32   %f53, [%r18+60];
ld.shared.f32   %f54, [%r17+1920];
fma.rn.f32  %f55, %f54, %f53, %f52;
```

(4+1) loads for 4 FMA

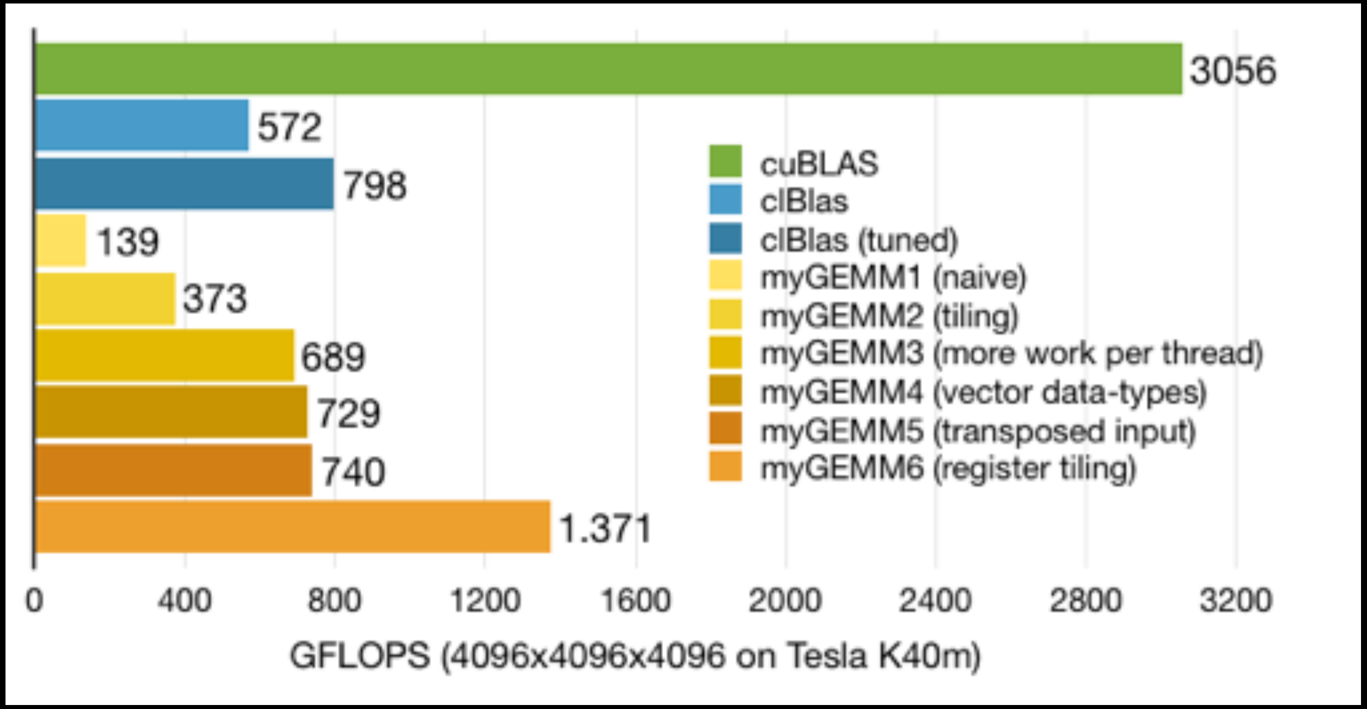


```
ld.shared.f32   %f82, [%r101+4];
ld.shared.f32   %f83, [%r102];
fma.rn.f32  %f91, %f83, %f82, %f67;
ld.shared.f32   %f84, [%r101+516];
fma.rn.f32  %f92, %f83, %f84, %f69;
ld.shared.f32   %f85, [%r101+1028];
fma.rn.f32  %f93, %f83, %f85, %f71;
ld.shared.f32   %f86, [%r101+1540];
fma.rn.f32  %f94, %f83, %f86, %f73;
```



```
// 4 + 1 register storage
float Areg;
float Breg[4];

// Cache the 4 values of Bsub in registers
for (int wn = 0; wn < 4; ++wn)
{
    int col = tidn + wn * 8;
    Breg[wn] = Bsub[col][k];
}

// Perform the 4 * 4 computations
for (int wm = 0; wm < 4; ++wm)
{
    int row = tidm + wm * 8;
    Areg = Asub[k][row];
    for (int wn = 0; wn < 4; ++wn)
    {
        acc[wm][wn] += Areg * Breg[wn];
    }
}
```

(4+4) loads for 4*4 FMA



# Agenda

1. Intro GPU architecture
2. Intro GPU programming model
3. CUDA/OpenCL by example: matrix-multiplication
4. **C++11 ❤️ GPU → SyCL**

# CUDA and OpenCL are not ideal

Drawbacks of CUDA

1. Vendor specific
2. Requires special compiler
3. Some boilerplate code
4. Difficult to debug
5. …

Drawbacks of OpenCL

1. Kernel source as string
2. C-API, not C++
3. Lots of boilerplate code
4. Even more difficult to debug
5. …

# Some alternatives to CUDA/OpenCL

| | C++ host API | Custom kernels | Inter-op with OpenCL | Method |
|---|---|---|---|---|
| **Bolt/Thrust** | ✔ | X | X | Parallel STL library |
| **Boost.Compute** | ✔ | ✔ | ✔ | Parallel STL + custom kernels |
| **OpenMP 4 / OpenACC** | ✔ | ✔ | X | Pragma directives |
| **C++AMP** | ✔ | ✔ | X | Kernel as lambda |
| **SyCL** | ✔ | ✔ | ✔ | Kernel as lambda |

# A game of thrones

Khal’s sickle

OpenCL SyCL

Khaleesi’s spear

OpenCL SPIR

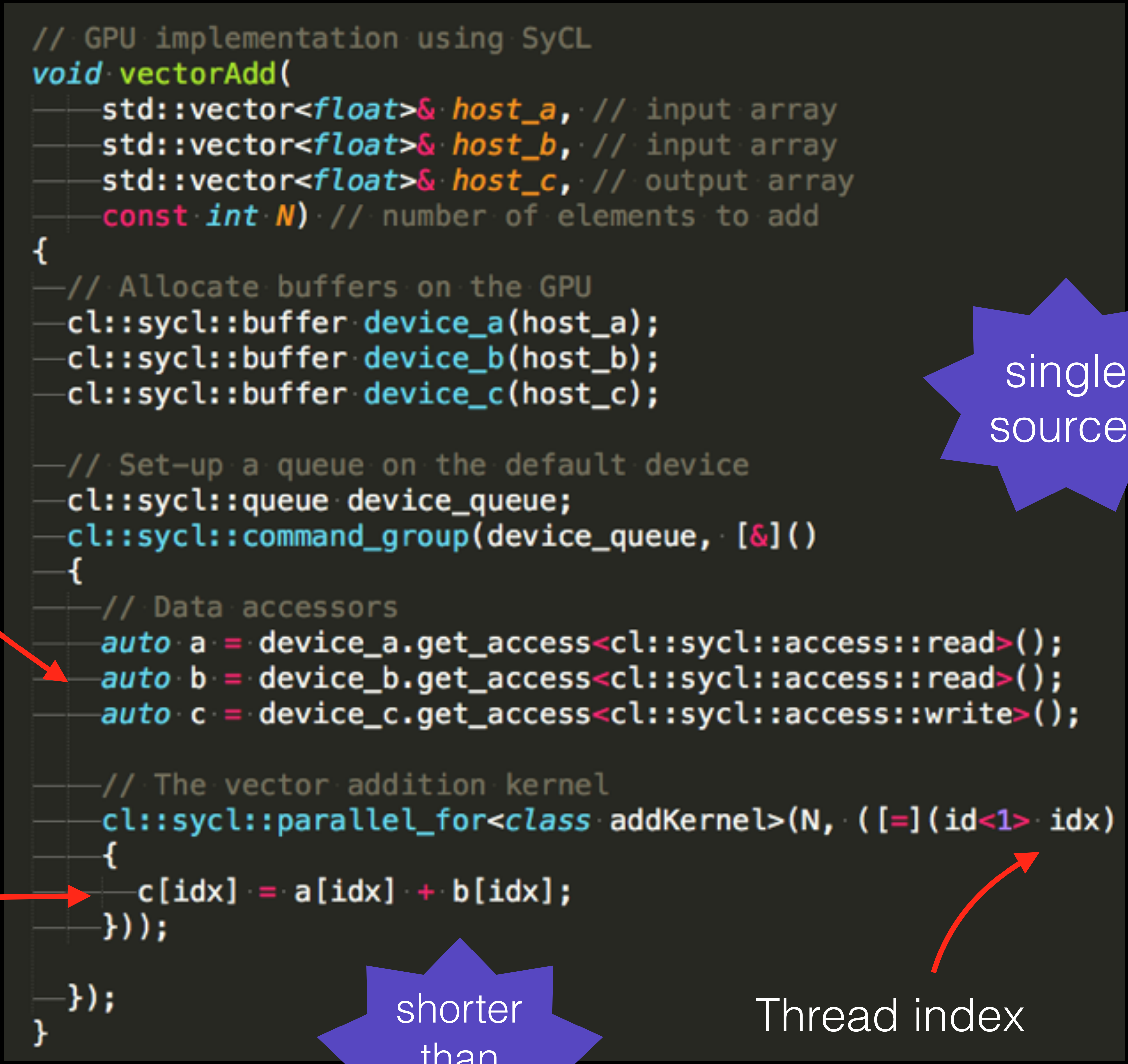
# Vector addition in SYCL

As before

Allocates and copies buffers (when needed)

valid C++

Device code



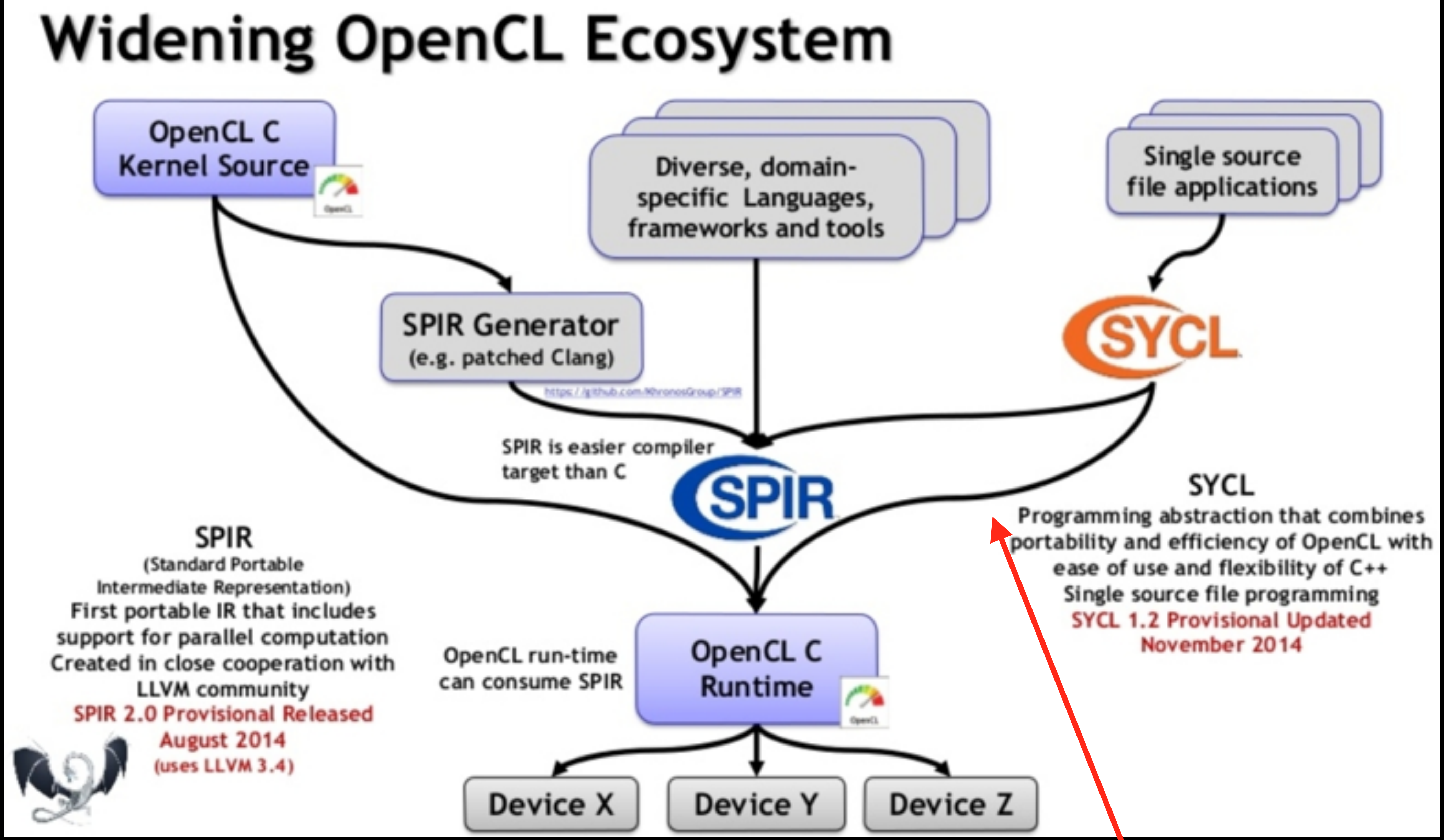
```
// GPU implementation using SyCL
void vectorAdd(
    std::vector<float>& host_a, // input array
    std::vector<float>& host_b, // input array
    std::vector<float>& host_c, // output array
    const int N) // number of elements to add
{
  // Allocate buffers on the GPU
  cl::sycl::buffer device_a(host_a);
  cl::sycl::buffer device_b(host_b);
  cl::sycl::buffer device_c(host_c);

  // Set-up a queue on the default device
  cl::sycl::queue device_queue;
  cl::sycl::command_group(device_queue, [&]()
  {
    // Data accessors
    auto a = device_a.get_access<cl::sycl::access::read>();
    auto b = device_b.get_access<cl::sycl::access::read>();
    auto c = device_c.get_access<cl::sycl::access::write>();

    // The vector addition kernel
    cl::sycl::parallel_for<class addKernel>(N, ([=](id<1> idx)
    {
      c[idx] = a[idx] + b[idx];
    }));

  });
}
```

single source!

shorter than CUDA!

Thread index

# The SYCL compute language



## Widening OpenCL Ecosystem

OpenCL C Kernel Source

Diverse, domain-specific Languages, frameworks and tools

Single source file applications

SPIR Generator (e.g. patched Clang)

https://github.com/KhronosGroup/SPIR

SYCL

SPIR is easier compiler target than C

SPIR

SPIR
(Standard Portable Intermediate Representation)
First portable IR that includes support for parallel computation
Created in close cooperation with LLVM community
SPIR 2.0 Provisional Released August 2014
(uses LLVM 3.4)

OpenCL run-time can consume SPIR

OpenCL C Runtime

SYCL
Programming abstraction that combines portability and efficiency of OpenCL with ease of use and flexibility of C++
Single source file programming
SYCL 1.2 Provisional Updated November 2014

Device X

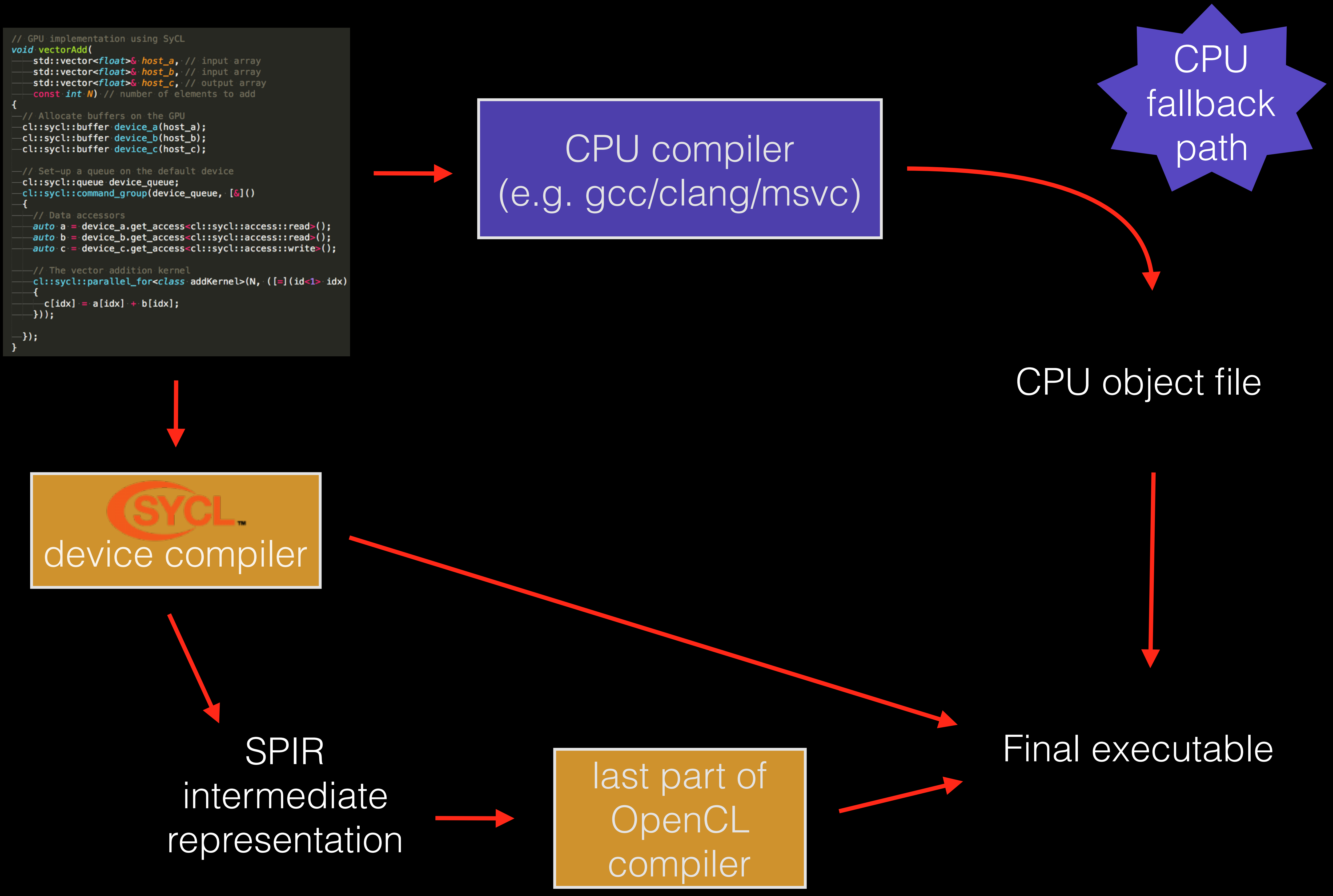
Device Y

Device Z

Runs on all existing OpenCL hardware

… as long as there is a compiler

# The SYCL compute language



```
// GPU implementation using SyCL
void vectorAdd(
        std::vector<float>& host_a, // input array
        std::vector<float>& host_b, // input array
        std::vector<float>& host_c, // output array
        const int N) // number of elements to add
{
    // Allocate buffers on the GPU
    cl::sycl::buffer device_a(host_a);
    cl::sycl::buffer device_b(host_b);
    cl::sycl::buffer device_c(host_c);

    // Set-up a queue on the default device
    cl::sycl::queue device_queue;
    cl::sycl::command_group(device_queue, [&]()
    {
        // Data accessors
        auto a = device_a.get_access<cl::sycl::access::read>();
        auto b = device_b.get_access<cl::sycl::access::read>();
        auto c = device_c.get_access<cl::sycl::access::write>();

        // The vector addition kernel
        cl::sycl::parallel_for<class addKernel>(N, ([=](id<1> idx)
        {
            c[idx] = a[idx] + b[idx];
        }));

    });
}
```

CPU compiler (e.g. gcc/clang/msvc)

CPU fallback path

CPU object file

SYCL device compiler

SPIR intermediate representation

last part of OpenCL compiler

Final executable

# GPU programming 101

1. Intro GPU architecture
2. Intro GPU programming model
3. CUDA/OpenCL by example
4. C++11 ❤️ GPU → SyCL

*Cedric Nugteren*

Amsterdam C++ meetup
2016 - 08 - 25