COALESCED GLOBAL MEMORY

LEARNING OBJECTIVES

- Learn about coalesced global memory access
- Learn about the performance impact
- Learn about row-major vs column-major
- Learn about SoA vs AoS

COALESCED GLOBAL MEMORY

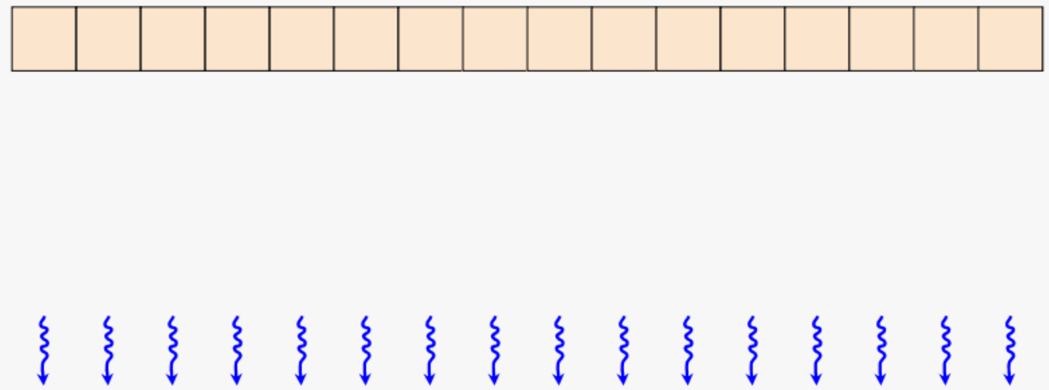
- Reading from and writing to global memory is generally very expensive.
- It often involves copying data across an off-chip bus.
    - This means you generally want to avoid unnecessary accesses.
- Memory access operations is done in chunks.
    - This means accessing data this is physically close together in memory is more efficient.

COALESCED GLOBAL MEMORY

```
float data[size];
```

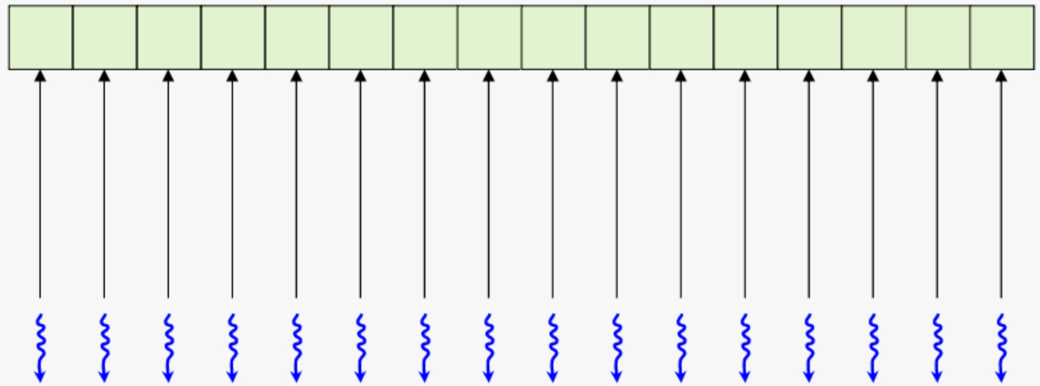
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId]);
```



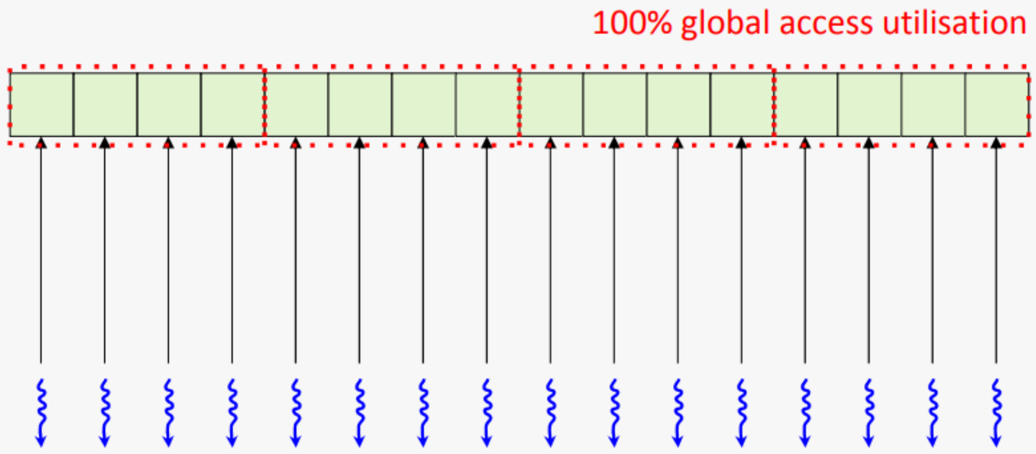
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId]);
```



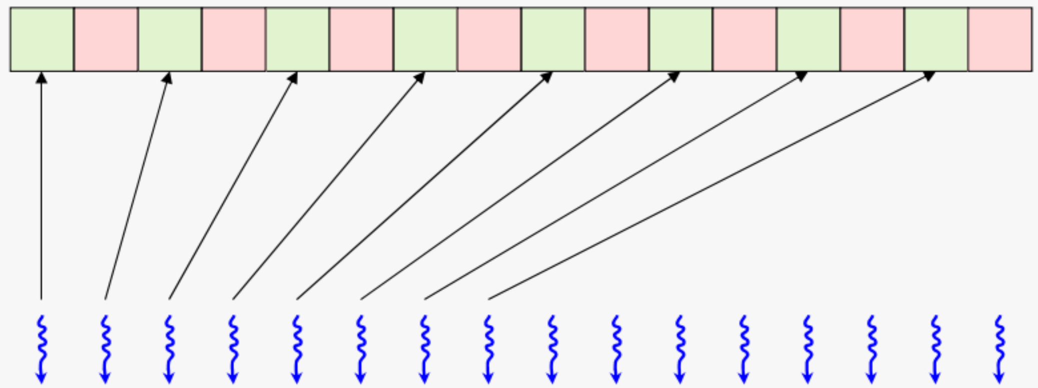
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId]);
```



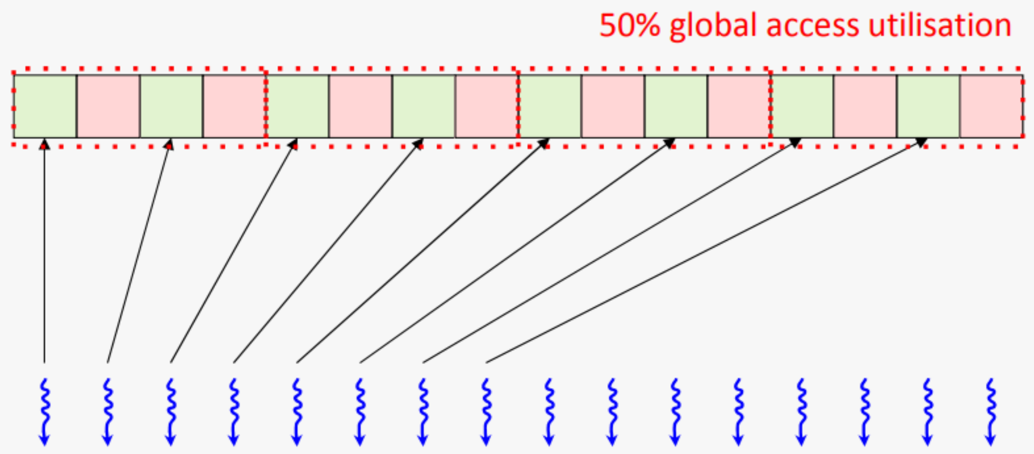
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId * 2]);
```



COALESCED GLOBAL MEMORY

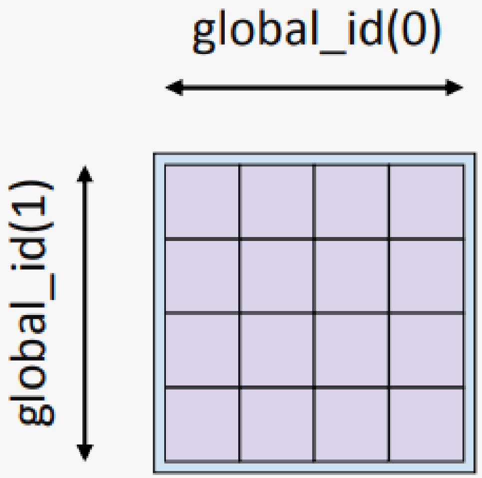
```
float data[size];

...

f(a[globalId * 2]);
```



ROW-MAJOR VS COLUMN-MAJOR

- Coalescing global memory access is particularly important when wording in multiple dimensions.
- This is because when doing so you have to convert from a position in 2d space to a linear memory space.
- There are two ways to do this; generally referred to as row-major and column-major.

ROW-MAJOR VS COLUMN-MAJOR

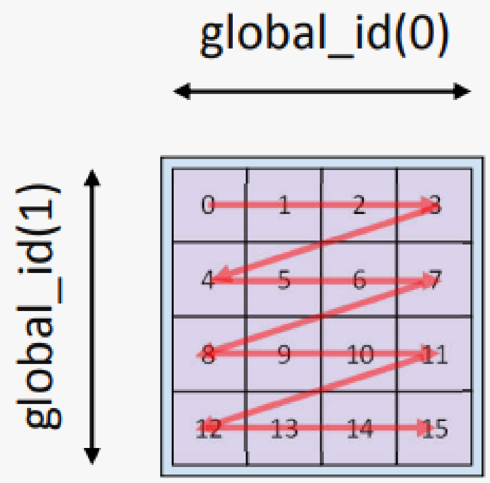


Row-major

```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id1 * 4) + id0;
a[linearId] = f();
```

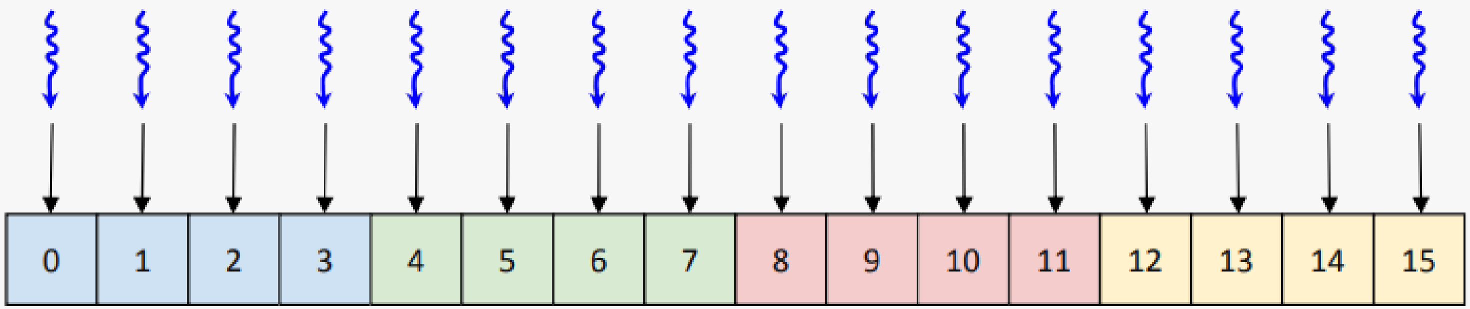
ROW-MAJOR VS COLUMN-MAJOR

Row-major

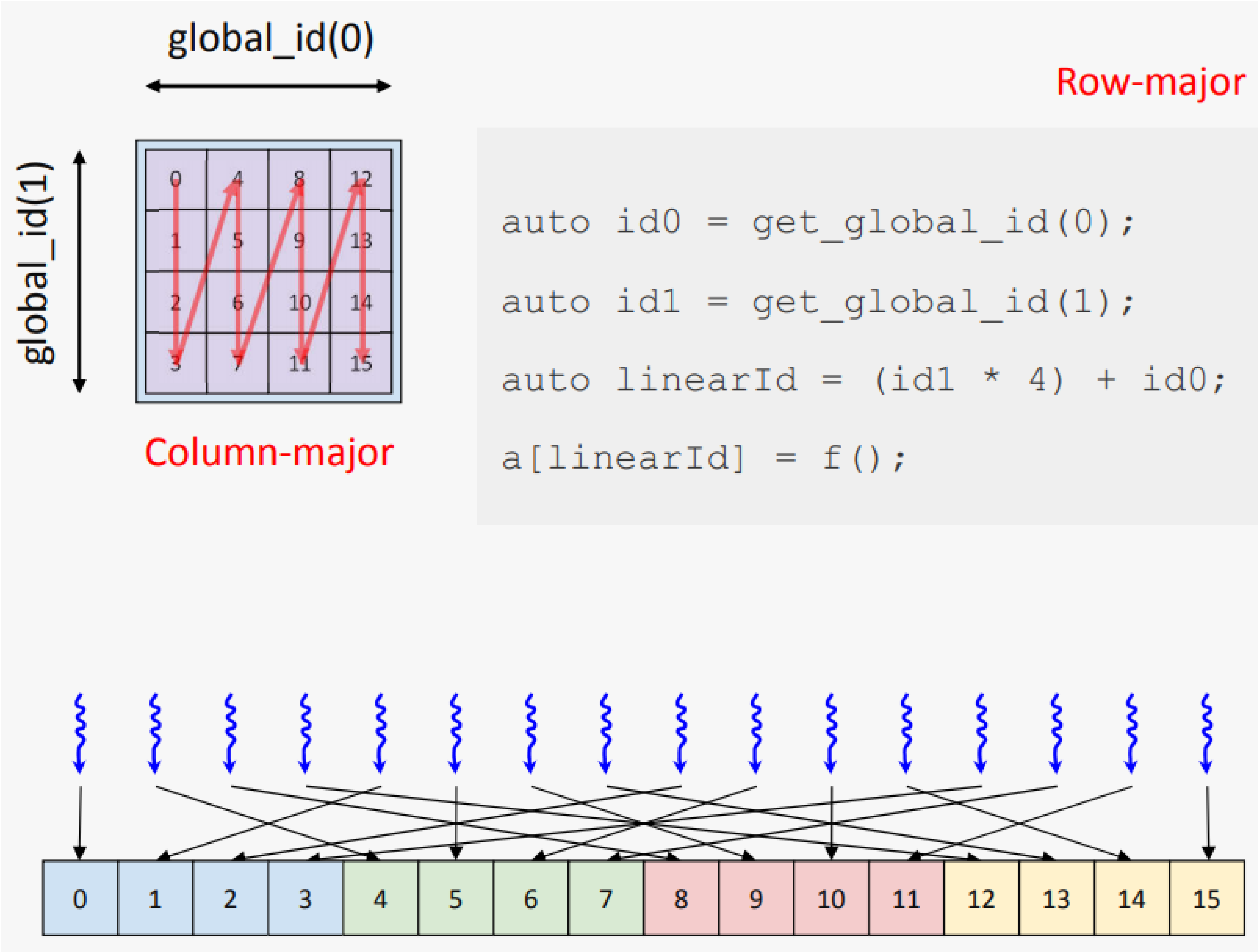


Row-major

```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id1 * 4) + id0;
a[linearId] = f();
```



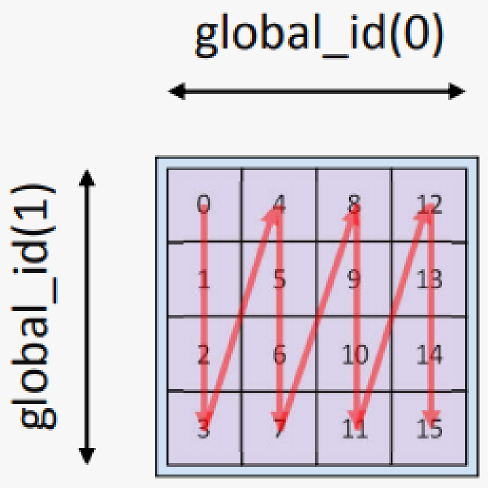
ROW-MAJOR VS COLUMN-MAJOR



global_id(0)
Row-major
global_id(1)
Column-major
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id1 * 4) + id0;
a[linearId] = f();
0 4 8 12
1 5 9 13
2 6 10 14
3 7 11 15
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

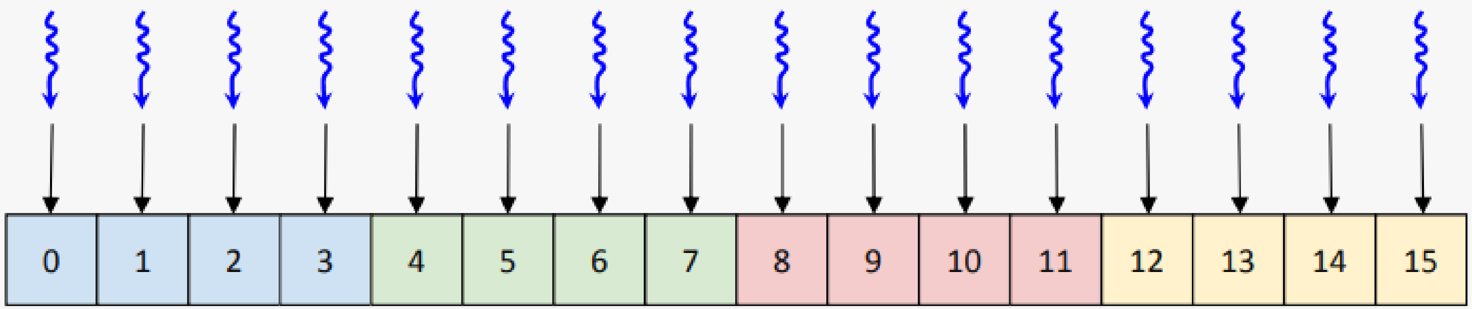
ROW-MAJOR VS COLUMN-MAJOR

Column-major

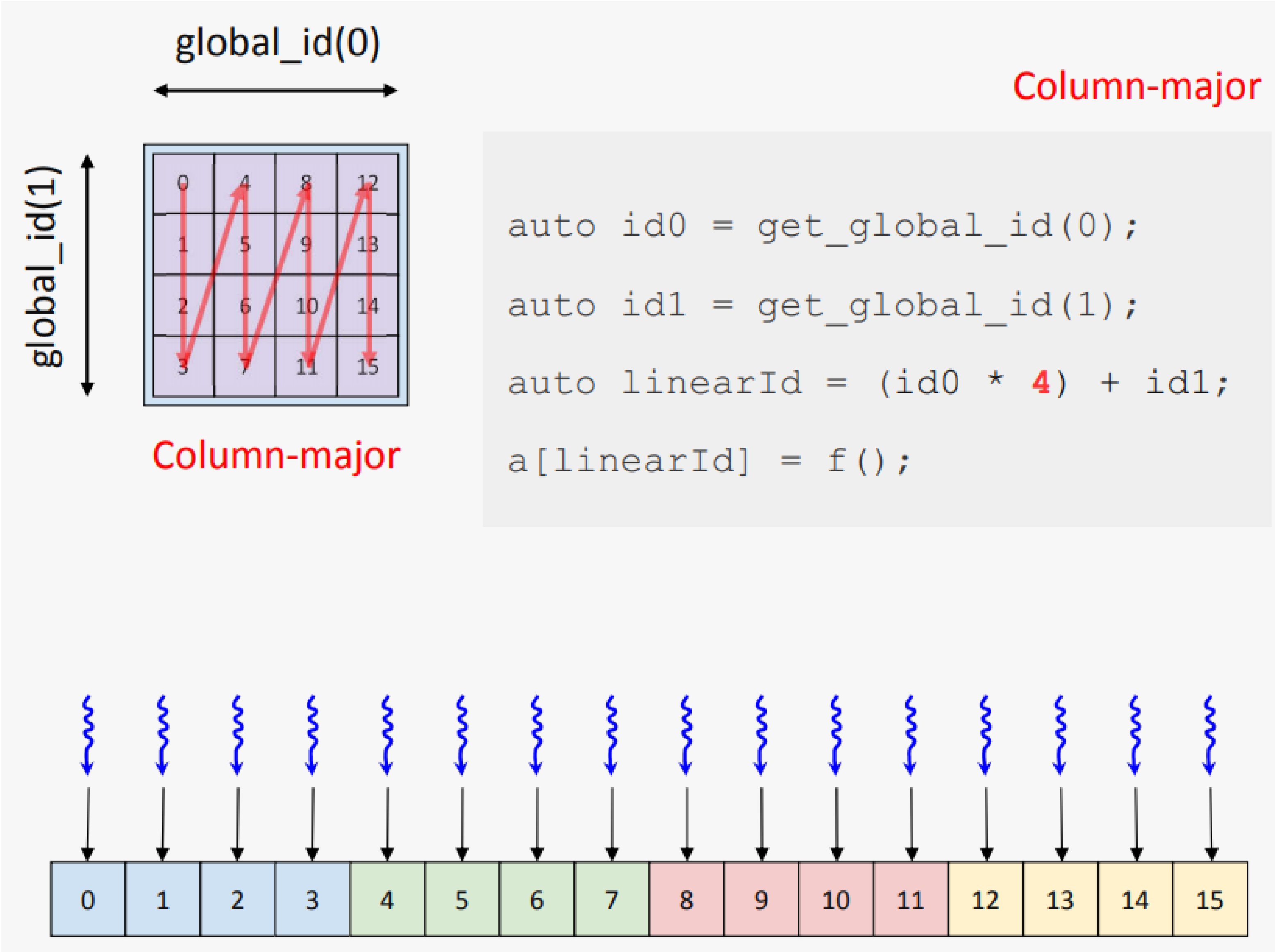


Column-major

```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id0 * 4) + id1;

a[linearId] = f();
```



ROW-MAJOR VS COLUMN-MAJOR

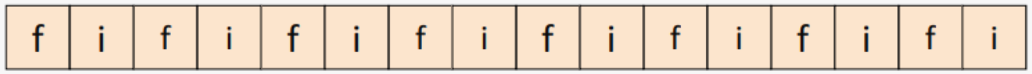

global_id(0)
Column-major
global_id(1)
0 4 8 12
1 5 9 13
2 6 10 14
3 7 11 15
Column-major
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id0 * 4) + id1;
a[linearId] = f();
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

AOS VS SOA

- Another area this is a factor is when composing data structures.
- It's often instinctive to have struct representing a collection of data and then have an array of this - often referred to as Array of Structs (AoS).
- But for data parallel architectures such as a GPU it's more efficient to have sequential elements of the same type stored contiguously in memory - often referred to as Struct of Arrays (SoA).

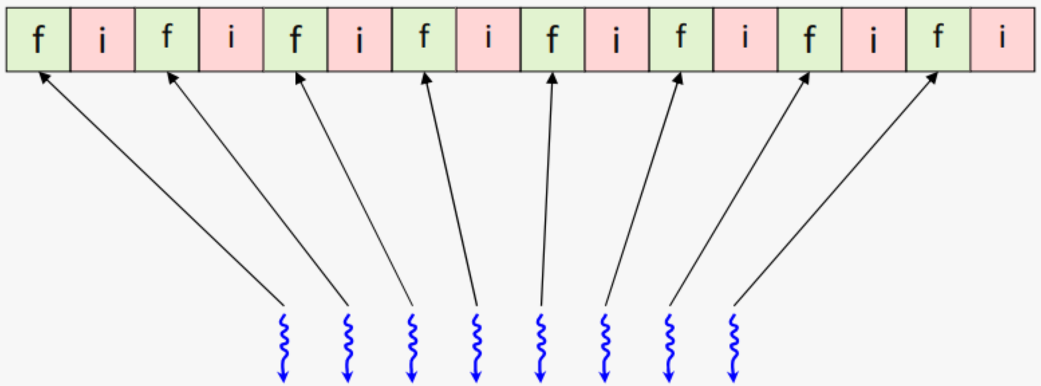
AOS VS SOA

```
struct str {
  float f;
  int i;
};

str data[size];
```



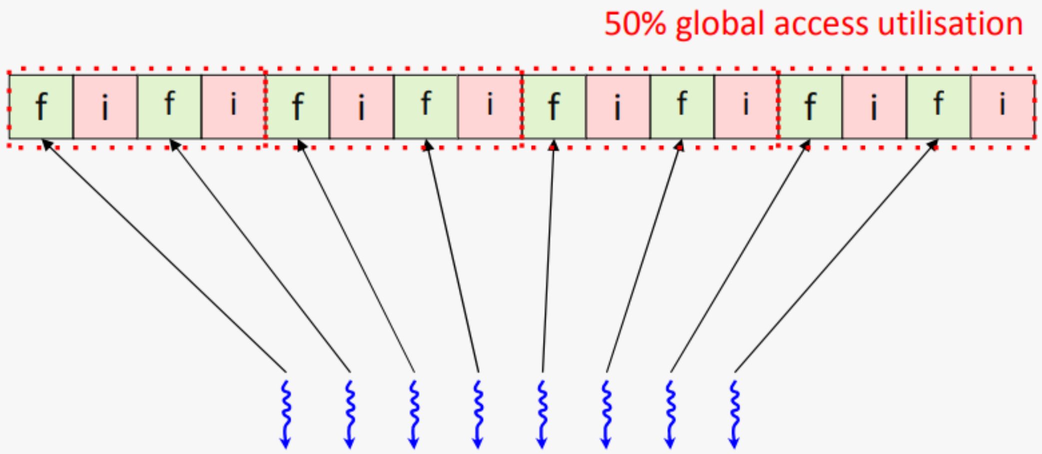
AOS VS SOA

```
struct str {
  float f;
  int i;
};

str data[size];

...

f(a[globalId].f);
```



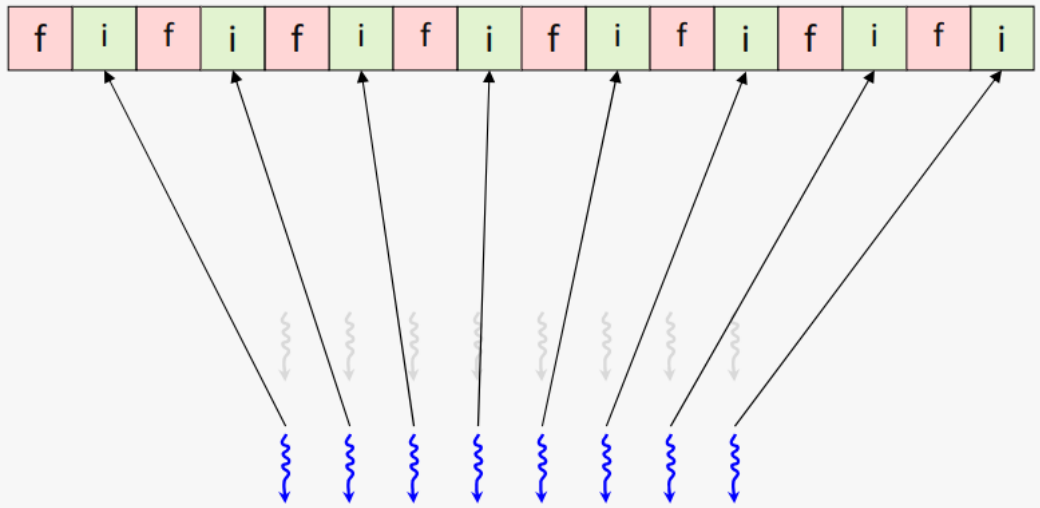
AOS VS SOA

```
struct str {
  float f;
  int i;
};

str data[size];

...

f(a[globalId].f);
```



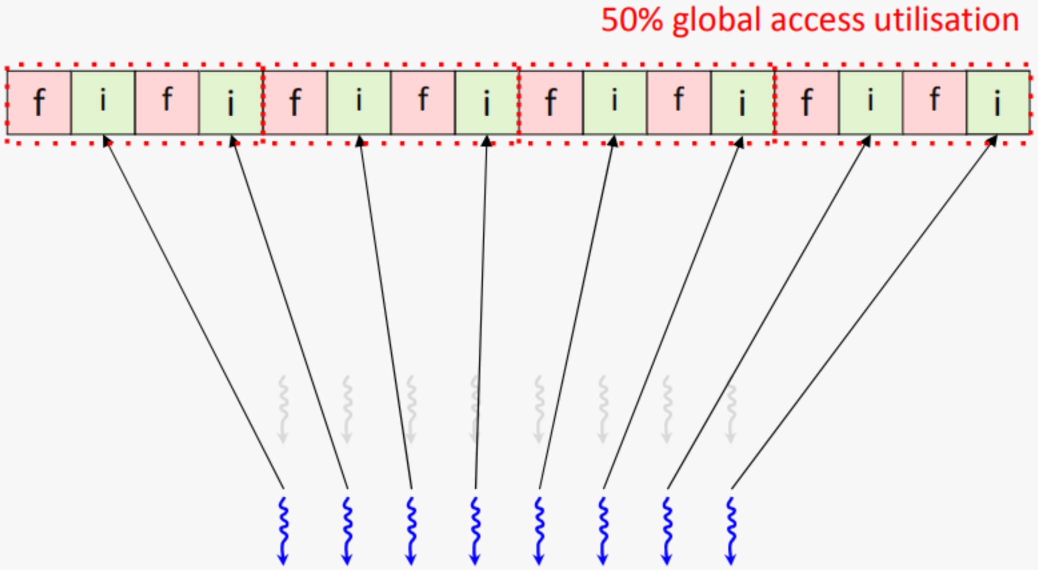
AOS VS SOA

```
struct str {
  float f;
  int i;
};

str data[size];

...



f(a[globalId].f);

f(a[globalId].i);
```



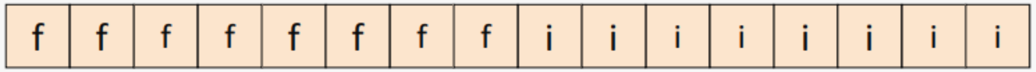
AOS VS SOA

```
struct str {
  float f;
  int i;
};

str data[size];

...


f(a[globalId].f);

f(a[globalId].i);
```



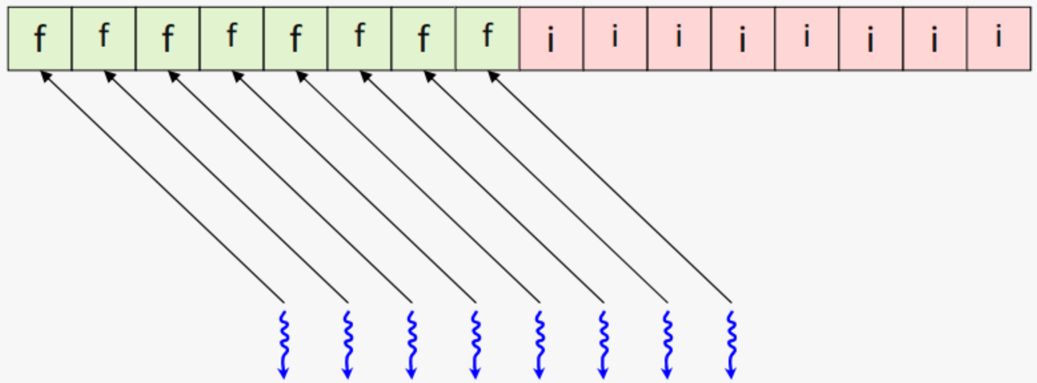
AOS VS SOA

```
struct str {
  float fs[size];
  int is[size];
};

str data;
```



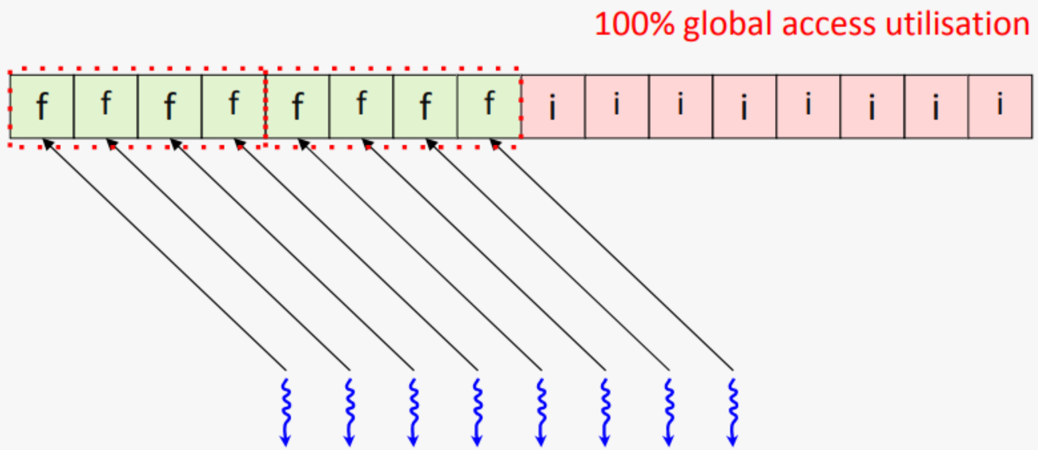
AOS VS SOA

```
struct str {
  float fs[size];
  int is[size];
};

str data;

...


f(a[0].fs[globalId]);
```



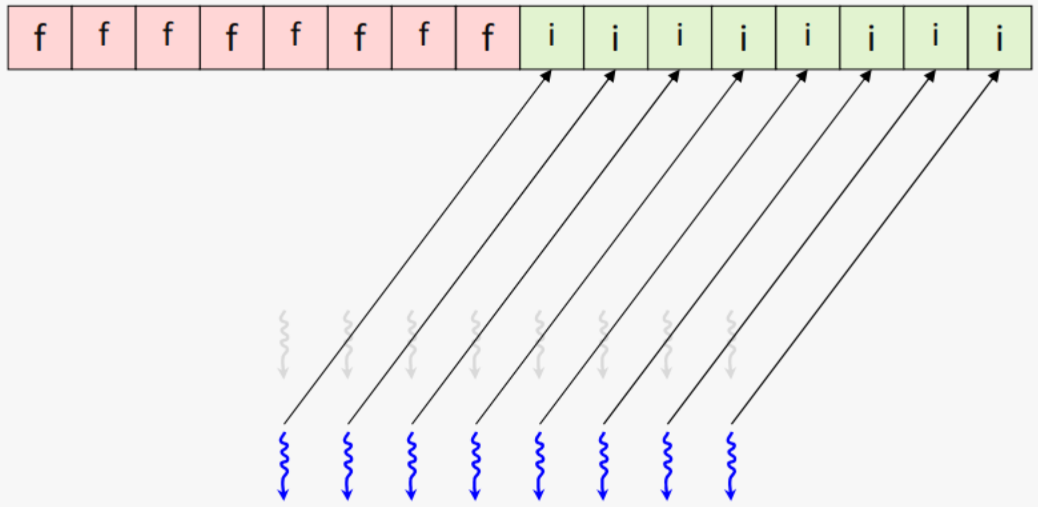
AOS VS SOA

```
struct str {
  float fs[size];
  int is[size];
};

str data;

...


f(a[0].fs[globalId]);
```



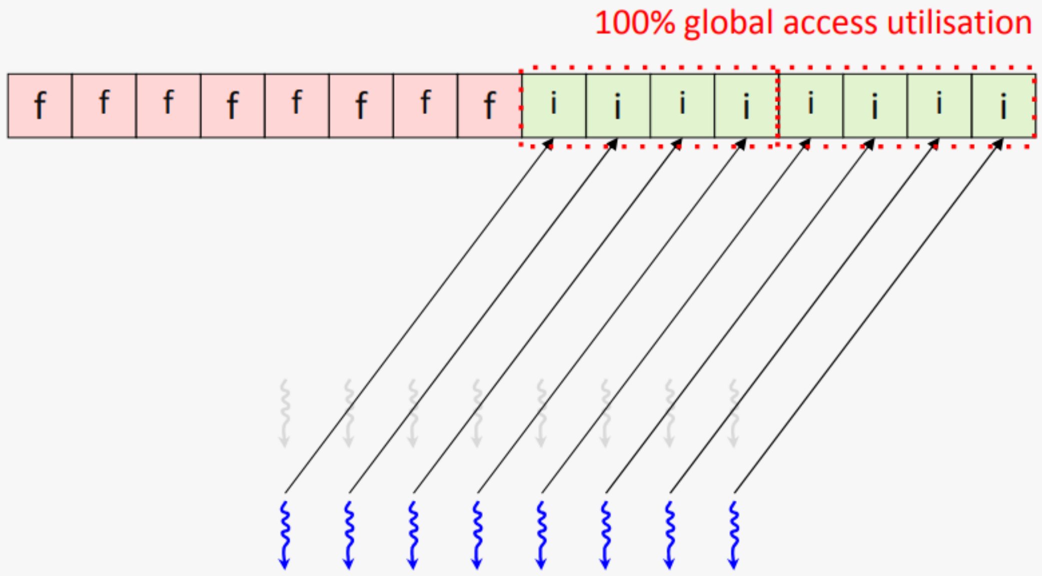
AOS VS SOA

```
struct str {
  float fs[size];
  int is[size];
};

str data;

...

f(a[0].fs[globalId]);

f(a[0].is[globalId]);
```



AOS VS SOA

```
struct str {
  float fs[size];
  int is[size];
};

str data;

...

f(a[0].fs[globalId]);

f(a[0].is[globalId]);
```



COALESCED IMAGE CONVOLUTION PERFORMANCE

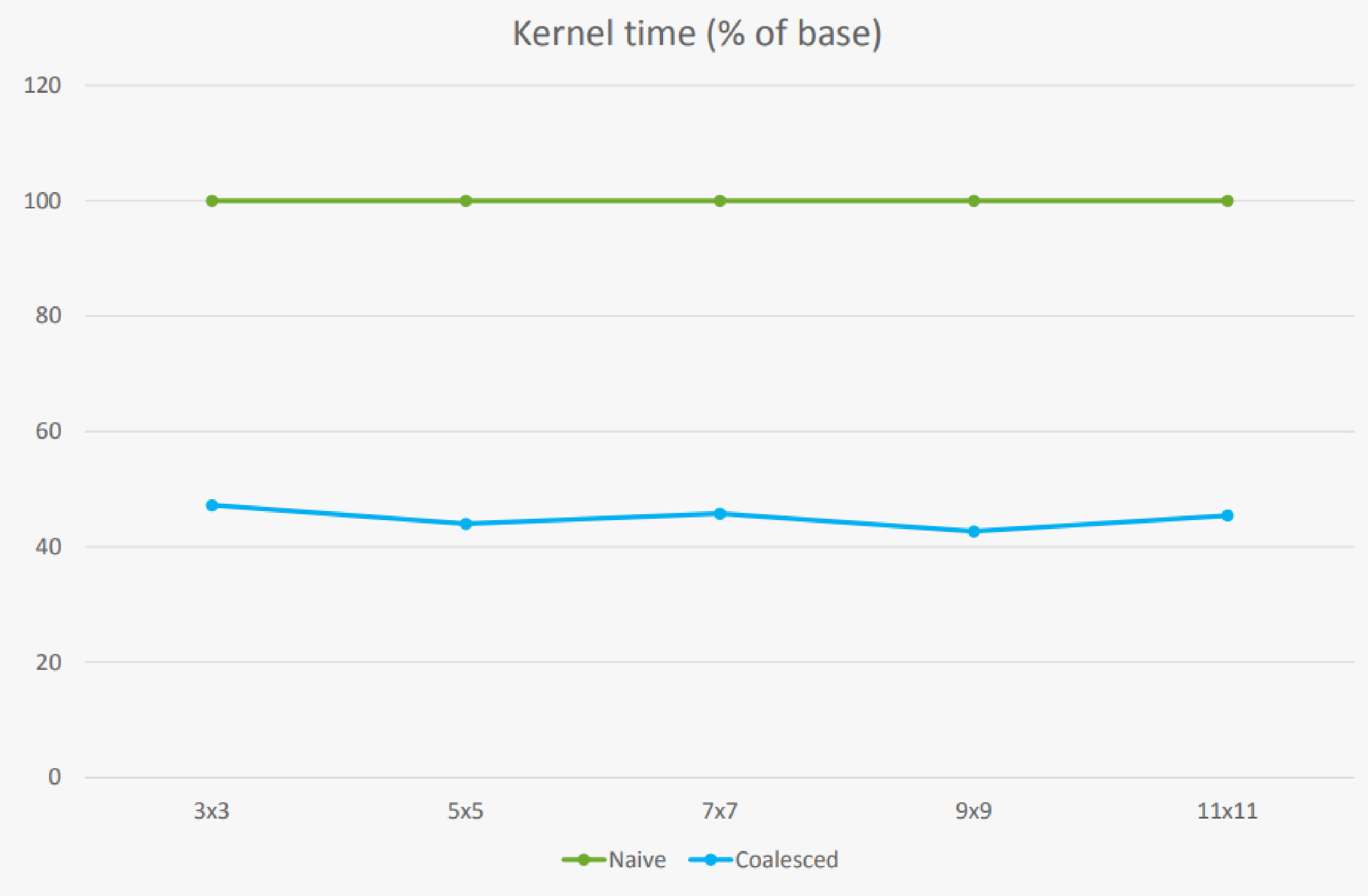


Image conv

512x512 source image

Intel HD Graphics 530

QUESTIONS

EXERCISE

Code_Exercises/Exercise_16_Coalesced_Global_Memory/source

Try inverting the dimensions when calculating the linear address in memory and measure the performance.