Australian National University

# OpenCL Kernel Programming

# OpenCL C for Compute Kernels

- Derived from **ISO C99**
  - ***Restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 **are** supported (**`#include, #if`** etc.)
- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - **`convert_type<_sat><_roundingmode>`**
  - Image types:
    - **`image2d_t, image3d_t`** and **`sampler_t`**

# OpenCL C for Compute Kernels

- Built-in functions — ***mandatory***
  - Work-Item functions, **math.h**, read/write images
  - Relational, geometric functions, synchronization functions
  - printf (OpenCL v1.2 or later)
- Built-in functions — ***optional "extensions"***
  - Double precision, atomics to global and local memory
  - Selection of rounding modes, writes to image3d_t surface

# OpenCL C Language Highlights

- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other (non kernel) device-side functions
- Address space qualifiers
  - `__global, __local, __constant, __private`
  - Pointer kernel arguments **must** have an address space qualifier
- Work-item functions
  - `get_global_id(), get_local_id(), get_group_id()` etc.
- Synchronization functions
  - **Barriers** - all work-items within a work-group **must** execute the barrier function before any work-item can continue
  - **Memory fences** - provides ordering between memory operations

# OpenCL C Language Restrictions

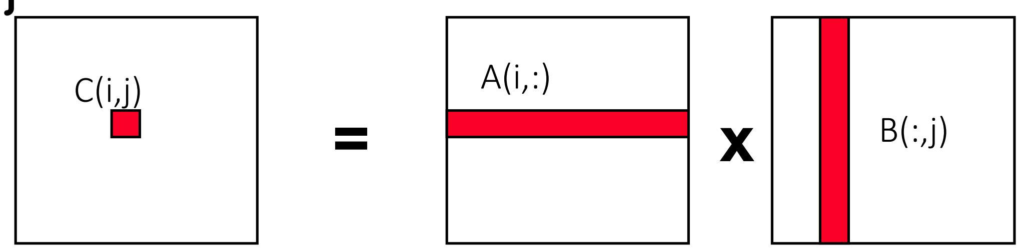
- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are ***not*** supported
- Variable length arrays and structures are ***not*** supported
- Recursion is ***not*** supported (yet!)
- Double types are **optional** in OpenCL v1.1, but the key word is reserved

  (note: most implementations support double)

# Matrix multiplication: sequential code

We calculate C=AB, where all three matrices are NxN

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```




Dot product of a row of A and a column of B for each element of C

# Matrix multiplication performance

- Serial C code on CPU (single core).

| Case | GFLOP/s | |
|---|---|---|
| | CPU | GPU |
| [redacted] | | |

Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz (36 cores total) using gcc v6.1.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

# Matrix multiplication: OpenCL kernel (1/2)

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }
        }
    }
}
```

Mark as a kernel function and specify memory qualifiers

# Matrix multiplication: OpenCL kernel (2/2)

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
        j = get_global_id(1);
            C[i*N+j] = 0.0f;
            for (k = 0; k < N; k++) {
                // C(i, j) = sum(over k) A(i,k) * B(k,j)
                C[i*N+j] += A[i*N+k] * B[k*N+j];
            }

}
```

Unroll the two outermost loops, instead setting loop indices i and j to be the global ids in the x and y direction

# Matrix multiplication: OpenCL kernel

```
__kernel void mat_mul(const int N, __global float *A,
                      __global float *B, __global float *C)
{
    int i, j, k;
    i = get_global_id(0);
    j = get_global_id(1);
    C[i*N+j] = 0.0f;
    for (k = 0; k < N; k++) {
        // C(i, j) = sum(over k) A(i,k) * B(k,j)
        C[i*N+j] += A[i*N+k] * B[k*N+j];
    }
}
```

Tidy up a bit

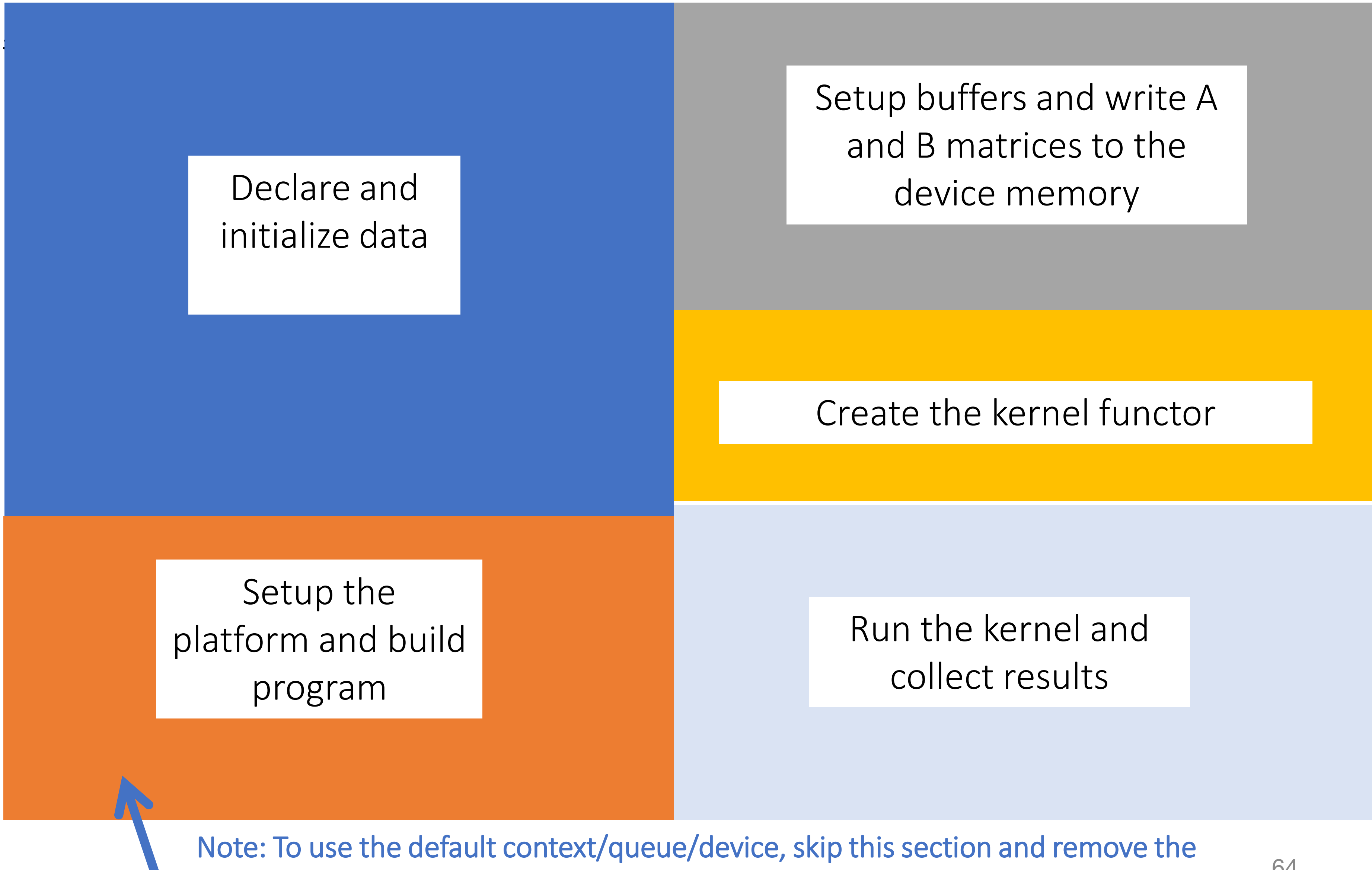
# Matrix multiplication: OpenCL kernel improved

Rearrange and use a local scalar for intermediate C element values (a common optimization in matrix multiplication functions)

```
__kernel void mmul(
    const int N,
    __global float *A,
    __global float *B,
    __global float *C)
{
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp = 0.0f;
    for (k = 0; k < N; k++)
      tmp += A[i*N+k]*B[k*N+j];

    C[i*N+j] = tmp;
}
```

# Matrix multiplication host program (C++ API)




Note: To use the default context/queue/device, skip this section and remove the references to context, queue and device.

# Matrix multiplication performance

- Matrices are stored in global memory.
- All the following results are from running C host code

| Case | GFLOP/s | |
|---|---|---|
| | CPU | GPU |
| [redacted] | | |
| C(i,j) per work-item, all global | 111.8 | 70.3 |

Device is NVIDIA® Tesla® P100 GPU with 56 compute units, 3,584 PEs, CUDA 9.1.84
Device is 2x Intel® Xeon® CPU, E5-2695 v4 @ 2.1GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.