Introduction to Parallel Processing

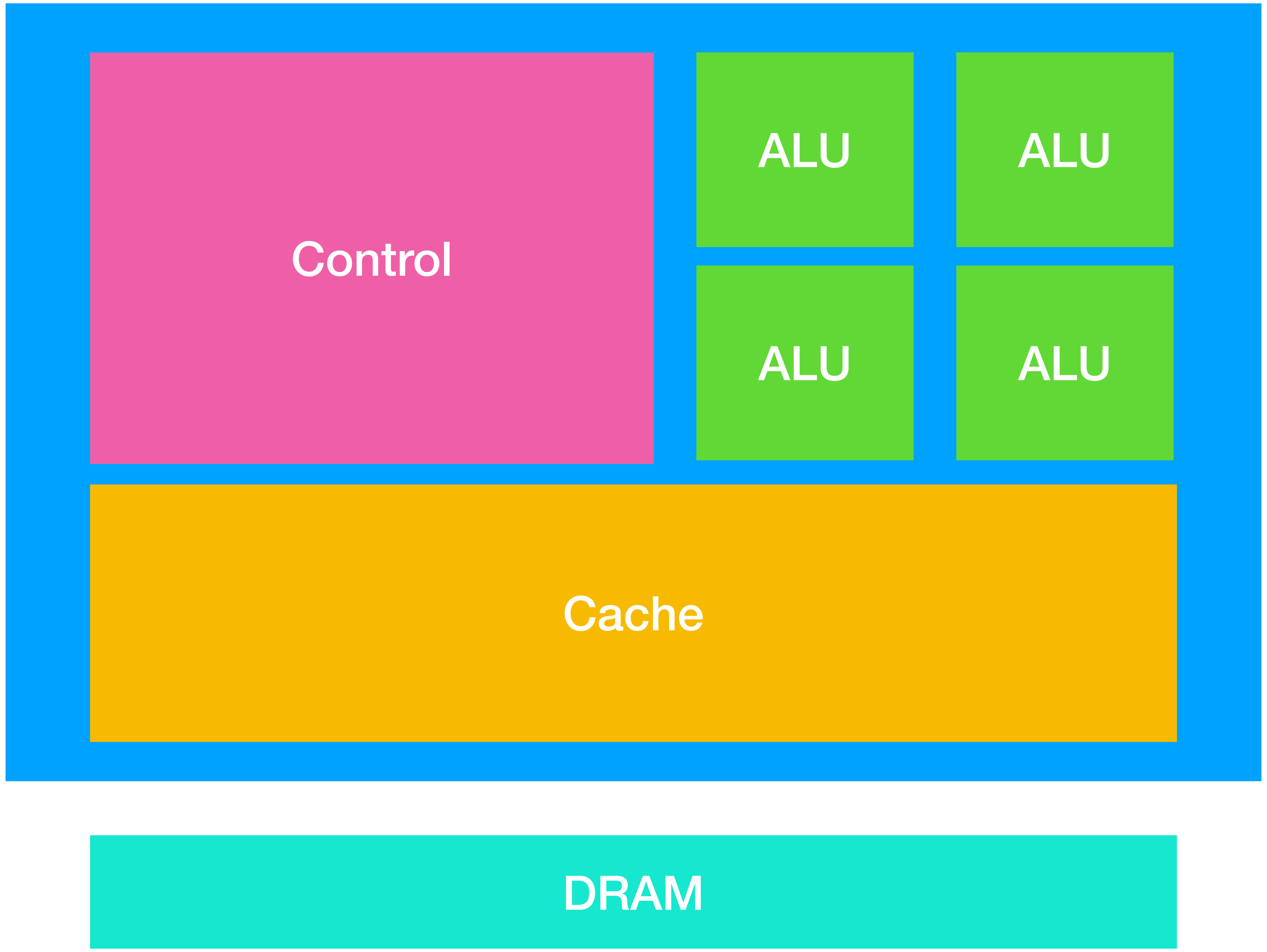
Lecture 25 : OpenMP

Professor Amanda Bienz

OpenMP: Easy Parallelism!

- Let’s step through an example, adding two lists together
- To have multiple threads each add part of these lists, we only need to add a single line to our code!
- **Caveat : getting efficient code, understanding if you have race conditions, etc is much more difficult**

Vectorization



Vectorization

- #pragma omp simd : directly before loop you want to vectorize

```c
// An example loop that my compiler is able to vectorize
// GCC can detect that this loop can be rewritten without dependencies
void loop(int n, float* x, float* y, float* z, int n_iter)
{
    for (int iter = 0; iter < n_iter; iter++)
    {
#pragma omp simd
        for (int i = 1; i < n; i++)
        {
            z[i] = x[i] * y[i] * z[i];
        }
    }
}
```

Vectorization

- Aligned : tells compiler this pointer is allocated with aligned data. Can pass alignment, or leave blank for default alignment. Not always necessary to vectorize code.

```
// An example loop that my compiler is able to vectorize
// GCC can detect that this loop can be rewritten without dependencies
void loop(int n, float* x, float* y, float* z, int n_iter)
{
    for (int iter = 0; iter < n_iter; iter++)
    {
#pragma omp simd aligned(x, y, z)
        for (int i = 1; i < n; i++)
        {
            z[i] = x[i] * y[i] * z[i];
        }
    }
}
```

Vectorization

- Safelen : maximum number of iterations that can be safely vectorized without dependencies

```
// An example loop that some compilers may vectorize (my compiler can)
void loop_maybe_vec(int n, float* x, float* y, float* z, int n_iter)
{
    for (int iter = 0; iter < n_iter; iter++)
    {
#pragma omp simd safelen(4)
        for (int i = 4; i < n; i++)
        {
            z[i] = x[i] * y[i] * z[i-4];
        }
    }
}
```

Vectorization

- Vectorized reduction :

```
double norm(int n, float* x)
{
    double sum = 0;
#pragma omp simd reduction(+:sum)
    for (int i = 0; i < n; i++)
        sum += (x[i]*x[i]);
    return sum;
}
```

Compiling Vectorized Code

- If any include issues, add this header:
  - #include <omp.h>
- When compiling, need to pass at least -O1 optimization level
- Pass -fopenmp-simd
- E.g. : gcc -O2 -o omp_test omp_test.c -fopenmp-simd

Hello World with OpenMP

```
void hello_world()
{
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    printf("Hello world from thread %d of %d\n", thread_id, num_threads);
}
}
```

- Add -fopenmp to your compile line
- May need to use gcc rather than compilers such as clang

Setting Number of Threads

- Command Line : export OMP_NUM_THREADS=4

- Within program :

```c
void hello_world()
{
#pragma omp parallel num_threads(4)
{

    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    printf("Hello world from thread %d of %d\n", thread_id, num_threads);
}
}
```

Setting Number of Threads

- Command Line : export OMP_NUM_THREADS=8
- Within program :

```
void hello_world()
{
omp_set_num_threads(8);
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
    printf("Hello world from thread %d of %d\n", thread_id, num_threads);
}
}
```

Setting Number of Threads

- Regardless of method for setting threads :
  - Tells program maximum number of threads possible to use (OpenMP could decide to use fewer)

Shared Memory OpenMP

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();
#pragma omp master
{
    printf("Hello world from thread %d of %d\n", thread_id, num_threads);
}
}
```

- Master : block of code only executed by master thread
- Single : block of code only executed by a single thread, usually whichever is first to arrive
- Critical : block of code executed by all threads, but only by a single thread at a time

Critical vs Atomic

- Critical section : OpenMP adds locks around this
- Atomic : followed by a single line of code, which can be updated atomically at the hardware level
    - Read, Write, or Update

Barrier

- #pragma omp barrier : Creates a synchronization point
- Many OpenMP methods have implicit barriers as well
  - #pragma omp for : splits for loop evenly across threads, adds an implicit barrier at end
- `nowait` : tells OpenMP to remove this barrier
  - #pragma omp for nowait

Shared Memory OpenMP

- Easiest way to parallelize : #pragma omp parallel for
  - Splits for loop across threads

```
void matmult(double* A, double* B, double* C, int n)
{
#pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            dot_product(A, B, C, i, j, n);
        }
    }

}
```

Shared Memory OpenMP

- Always parallelize outermost loop. Overhead to initializing threads

```
void matmult(double* A, double* B, double* C, int n)
{
    for (int i = 0; i < n; i++)
    {
#pragma omp parallel for
        for (int j = 0; j < n; j++)
        {
            dot_product(A, B, C, i, j, n);
        }
    }
}
```

Shared Memory OpenMP

- If uneven amount of work per thread (or more threads than number of outer loops) can collapse loops

```
void matmult(double* A, double* B, double* C, int n)
{
#pragma omp parallel for collapse(2)
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            dot_product(A, B, C, i, j, n);
        }
    }

}
```

Shared and Private Variables

- Shared variables : all threads should access this variable
  - Example : reading from matrix A, all processes need to be able to share the matrix A

```
void matmult(double* A, double* B, double* C, int n)
{
#pragma omp parallel for num_threads(4) shared(A, B, C)
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            dot_product(A, B, C, i, j, n);
        }
    }

}
```

Shared and Private Variables

- Private variables : each process holds its own copy!
  - Do not want matrix A to be private (each thread will copy entirety of A)
  - Typically, for variables you are writing to
  - Only for variables declared before #pragma (j is not private)

```c
// Out Of Order Matmult
void matmult_oor(double* A, double* B, double* C, int n)
{
    double val;
#pragma omp parallel for num_threads(4) shared(A, B, C) private(val)
    for (int i = 0; i < n; i++)
    {
        for (int k = 0; k < n; k++)
            C[i*n+k] = 0;

        for (int j = 0; j < n; j++)
        {
            val = A[i*n+j];
            for (int k = 0; k < n; k++)
            {
                C[i*n+k] += val*B[j*n+k];
            }
        }
    }
}
```

Reductions

- Similar to omp simd reduction, can perform reduction in parallel

```c
double sum(double* A, int n)
{
    double s = 0;
#pragma omp parallel for reduction(+:s)
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            s += A[i*n+j];
            printf("%e\n", A[i*n+j]);
        }
    }
    return s;
}
```