codeplay®

THE HETEROGENEOUS SYSTEMS EXPERTS

Optimizing GEMM for manycore architectures

Goran Flegar

Universidad Jaume I, Spain
flegar@uji.es

SYCL BLAS team

General matrix-matrix product (GEMM)

$$C = \alpha \, \mathrm{op}_1(A) \, \mathrm{op}_2(B) + \beta C$$

- op_i is either identity or transpose
- α and β are scalars
- $\mathrm{op}_1(A)$ is m-by-k, $\mathrm{op}_2(B)$ is k-by-n, C is m-by-n *(column major storage)*

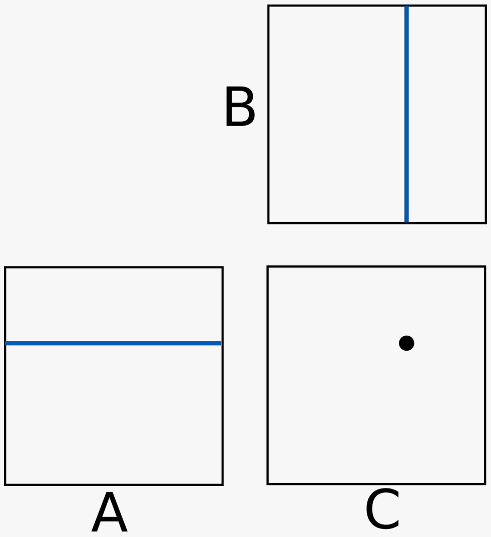
General matrix-matrix product (GEMM)

$$C = \alpha \, \mathrm{op}_1(A) \, \mathrm{op}_2(B) + \beta C$$

- op_i is either identity or transpose
- α and β are scalars
- $\mathrm{op}_1(A)$ is *m*-by-*k*, $\mathrm{op}_2(B)$ is *k*-by-*n*, C is *m*-by-*n (column major storage)*

In this talk (for simplicity): $C = AB$

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$



Naive implementation

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$

Map one work item to each element of c_{ij} and loop over $a_{i:}$ and $b_{:j}$.

Naive implementation

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$

AMD R9 Nano

8.19 Tflop/s peak performance

512 GB/s (128 Gfloat/s) bandwidth

Map one work item to each element of c_{ij} and loop over $a_{i:}$ and $b_{:j}$.

→ **~200 Gflop/s**

WHY?

4096-by-4096 matrices

Naive implementation

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$

AMD R9 Nano
8.19 Tflop/s peak performance
512 GB/s (128 Gfloat/s) bandwidth

SYCL ComputeCpp™

Map one work item to each element of c_{ij} and loop over $a_{i:}$ and $b_{:j}$.

~200 Gflop/s

WHY?

4096-by-4096 matrices

Each work item:

- $2k$ operations
- on $2k$ data elements

Memory bounded kernel!

Naive implementation

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$

Map one work item to each element of c_{ij} and loop over $a_{i:}$ and $b_{:j}$.

AMD R9 Nano

8.19 Tflop/s peak performance

512 GB/s (128 Gfloat/s) bandwidth

SYCL ComputeCpp™

~200 Gflop/s

WHY?

4096-by-4096 matrices

Each work item:

- $2k$ operations
- on $2k$ data elements

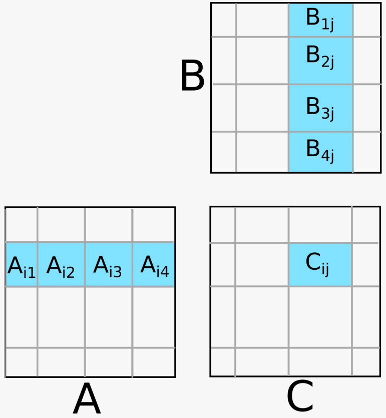
Memory bounded kernel!

Need to reuse data to “escape” memory bandwidth barrier.

8192 : 128 = 64 : 1

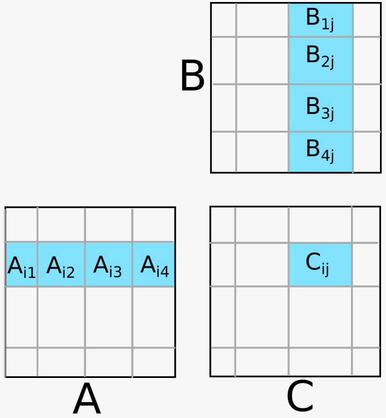
* Need at least 64 operations for each float fetched!

Block matrix multiplication



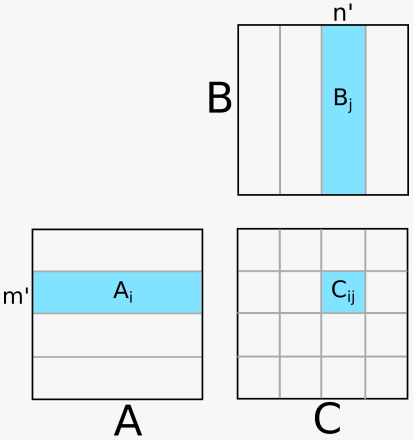
$$C_{ij} = \sum_{l=1}^{K} A_{il} B_{lj}$$

Block matrix multiplication



$$C_{ij} = \sum_{l=1}^{K} A_{il} B_{lj}$$

Special case: panel multiplication

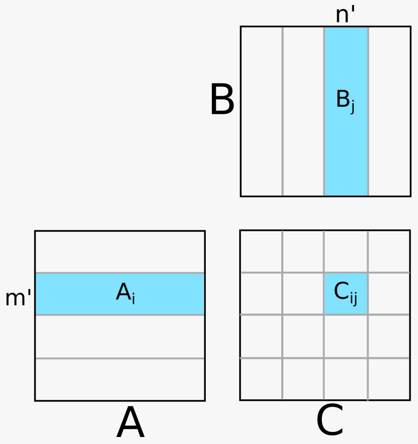


$$C_{ij} = A_i B_j$$

One work item per panel:

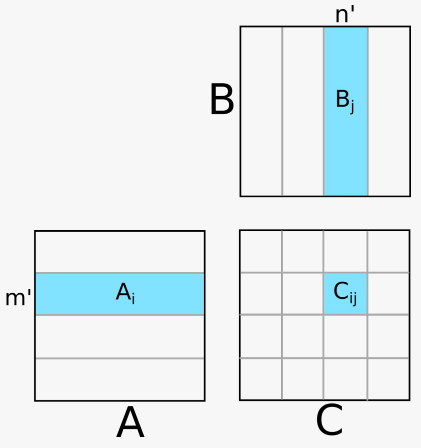
- *2m’n’k* operations
- on *m’k* + *kn’* + *m’n’* data elements

Maximizing data reuse

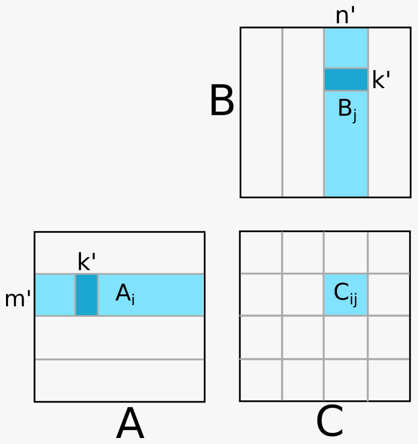


Cannot store the whole panel in caches / local memory / registers

Maximizing data reuse



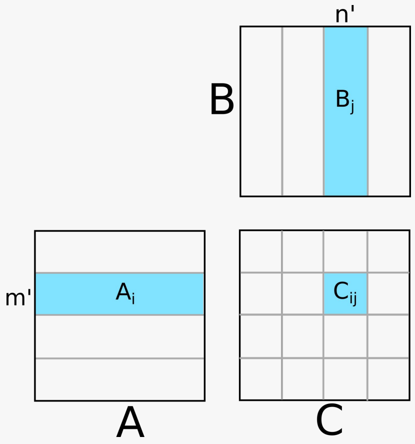
Cannot store the whole panel in caches / local memory / registers



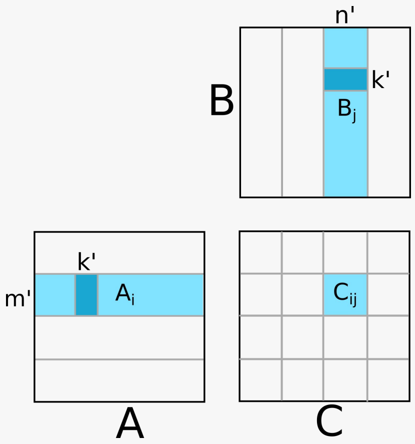
Instead break it into blocks

- Keep C_{ij} in registers
- Load a single *block* of *A* and *B*
  - m’k’ + k’n’ data
- Compute a small gemm with these blocks and add the result to C_{ij}
  - 2m’n’k’ operations
- Repeat the process for next block

Maximizing data reuse



Cannot store the whole panel in caches / local memory / registers



Instead break it into blocks

- Keep C_{ij} in registers
- Load a single *block* of *A* and *B*
  - m'k' + k'n' data
- Compute a small gemm with these blocks and add the result to C_{ij}
  - 2m'n'k' operations
- Repeat the process for next block

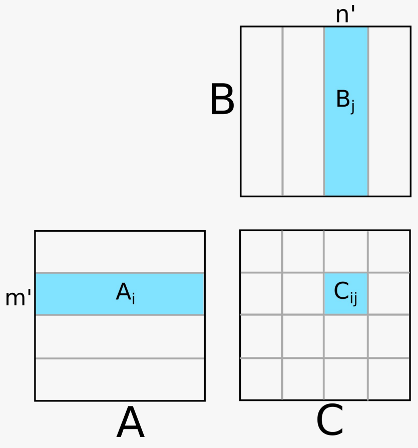
Data reuse:

$$\frac{2m'n'k'}{m'k'+n'k'} = \frac{2m'n'}{m'+n'}$$

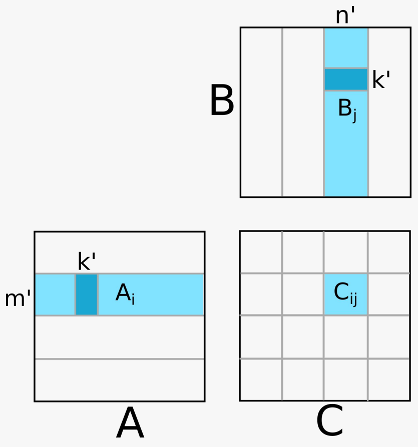
#registers:

$$m'n' + m'k' + k'n'$$

Maximizing data reuse



Cannot store the whole panel in caches / local memory / registers



Instead break it into blocks

Limited amount of registers:

- use k' as small as possible, keeping in mind good memory access
  - (k' = "cache line size")
- m' = n' is the best choice for constrained number of registers
  - "data reuse" = m'

- Keep C_{ij} in registers
- Load a single *block* of *A* and *B*
  - m'k' + k'n' data
- Compute a small gemm with these blocks and add the result to C_{ij}
  - 2m'n'k' operations
- Repeat the process for next block

Data reuse:

$$\frac{2m'n'k'}{m'k' + n'k'} = \frac{2m'n'}{m' + n'}$$

#registers:

$$m'n' + m'k' + k'n'$$

R9 Nano:

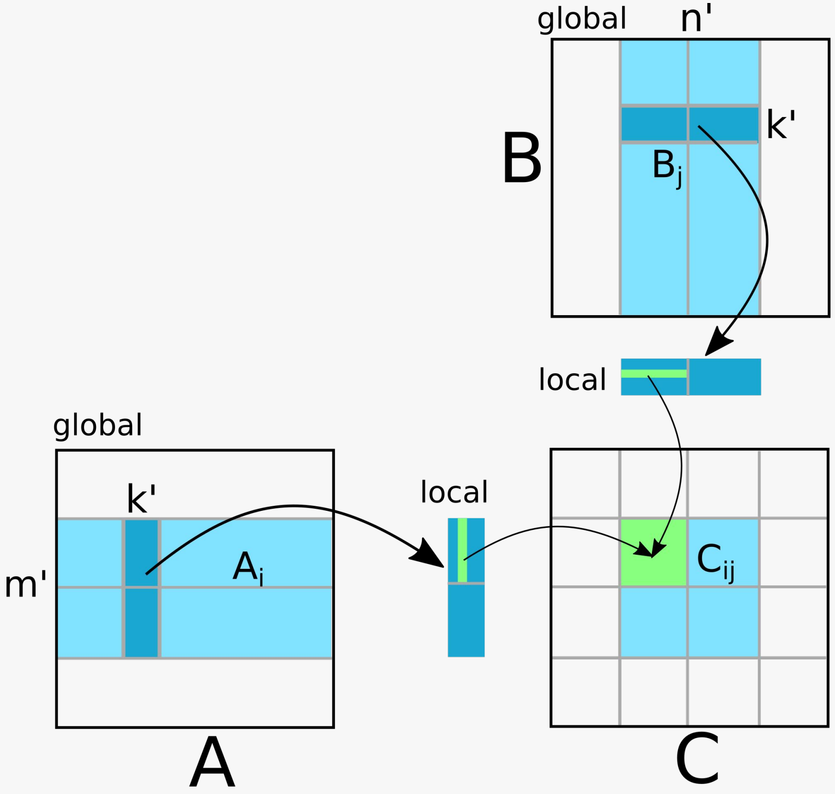
- "data reuse" = 8

Collaborate to increase effective data reuse

One work item has only a small amount of available registers.

- Combine the registers of entire workgroup to get more register space.

- Each work item stores only one sub-block of C_{ij}.
- All work items collaborate when reading to local memory.
- Each work item reads from local memory the part it needs.



Collaborate to increase effective data reuse

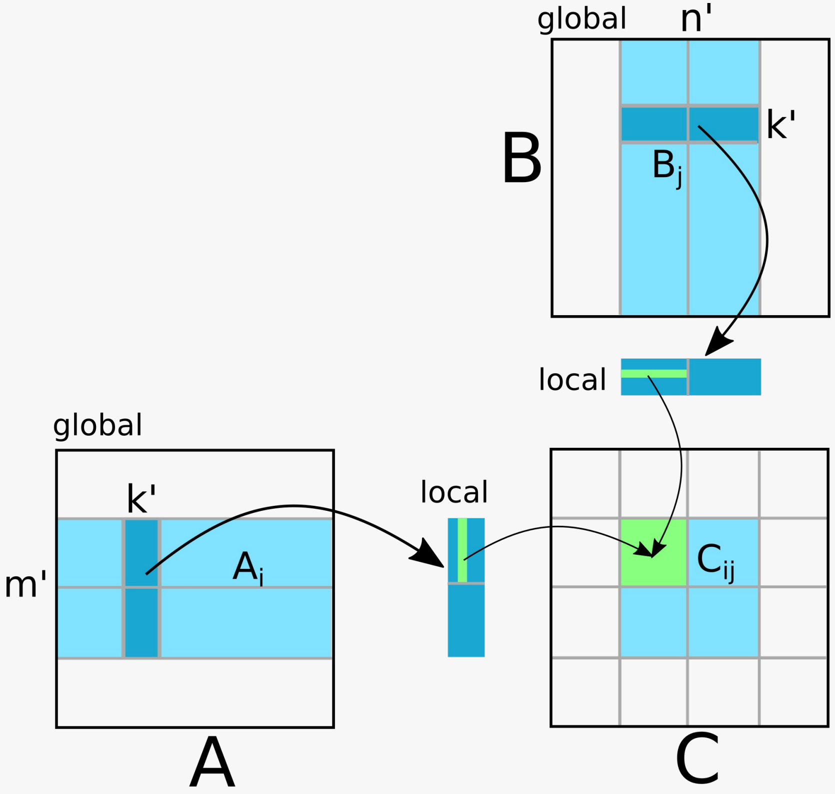
One work item has only a small amount of available registers.

- Combine the registers of entire workgroup to get more register space.

- Each work item stores only one sub-block of C_{ij}.
- All work items collaborate when reading to local memory.
- Each work item reads from local memory the part it needs.

R9 Nano:

- Work group size: 16x16 items
- "local data reuse" = 8
- "global data reuse" = 128



Further optimizations

Memory bandwidth no longer an issue.

Focus on decreasing the volume of “useless” arithmetic instructions.

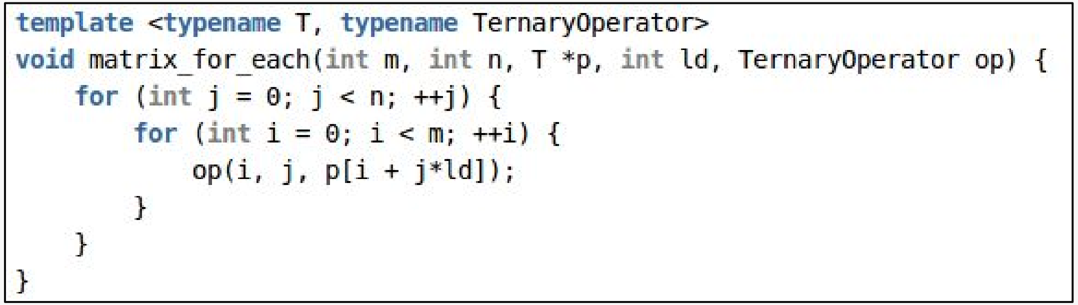
- Address calculation.
- Bound checking.

Further optimizations

Memory bandwidth no longer an issue.

Focus on decreasing the volume of “useless” arithmetic instructions.

- **Address calculation.**
- Bound checking.



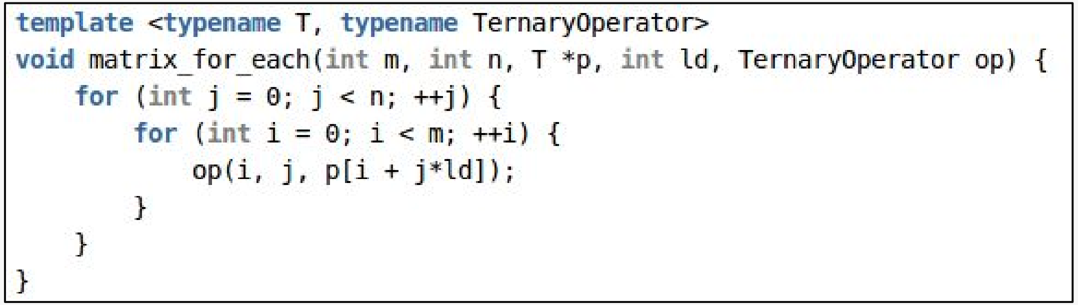
```
template <typename T, typename TernaryOperator>
void matrix_for_each(int m, int n, T *p, int ld, TernaryOperator op) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            op(i, j, p[i + j*ld]);
        }
    }
}
```

Further optimizations

Memory bandwidth no longer an issue.

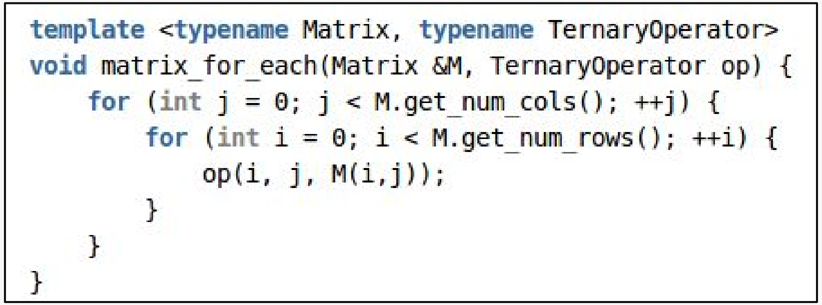
Focus on decreasing the volume of “useless” arithmetic instructions.

- **Address calculation.**
- Bound checking.



```
template <typename T, typename TernaryOperator>
void matrix_for_each(int m, int n, T *p, int ld, TernaryOperator op) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            op(i, j, p[i + j*ld]);
        }
    }
}
```

Introducing “matrix” abstractions might be tempting, but can have significant overhead.



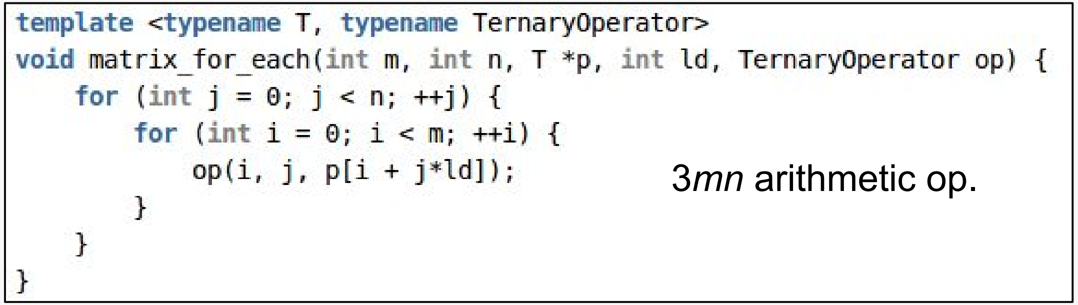
```
template <typename Matrix, typename TernaryOperator>
void matrix_for_each(Matrix &M, TernaryOperator op) {
    for (int j = 0; j < M.get_num_cols(); ++j) {
        for (int i = 0; i < M.get_num_rows(); ++i) {
            op(i, j, M(i,j));
        }
    }
}
```

Further optimizations

Memory bandwidth no longer an issue.

Focus on decreasing the volume of “useless” arithmetic instructions.

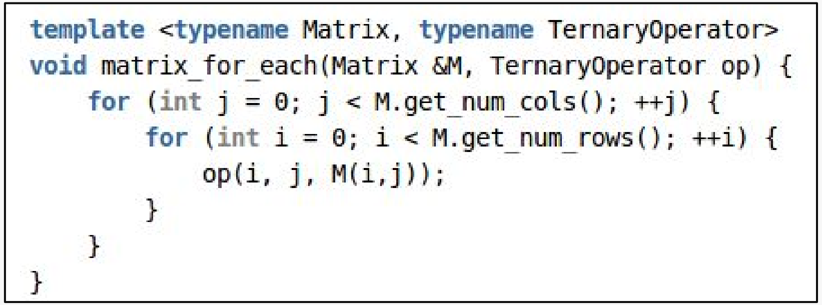
- **Address calculation.**
- Bound checking.



```
template <typename T, typename TernaryOperator>
void matrix_for_each(int m, int n, T *p, int ld, TernaryOperator op) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            op(i, j, p[i + j*ld]);
        }
    }
}
```

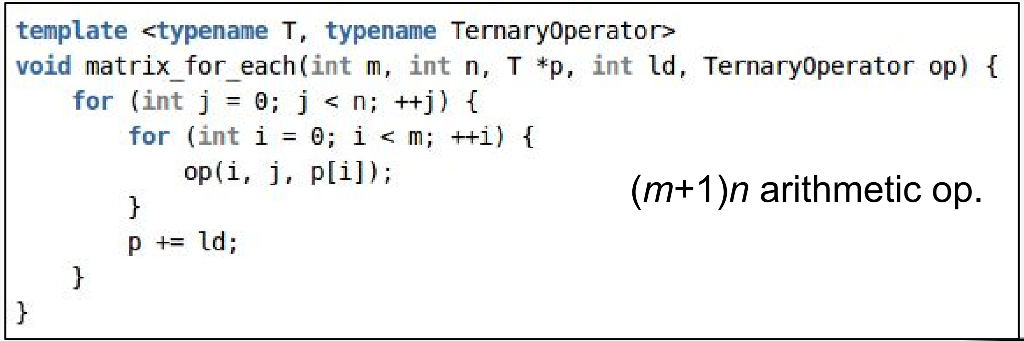
3mn arithmetic op.

Introducing “matrix” abstractions might be tempting, but can have significant overhead.



```
template <typename Matrix, typename TernaryOperator>
void matrix_for_each(Matrix &M, TernaryOperator op) {
    for (int j = 0; j < M.get_num_cols(); ++j) {
        for (int i = 0; i < M.get_num_rows(); ++i) {
            op(i, j, M(i,j));
        }
    }
}
```

Calculate partial addresses.



```
template <typename T, typename TernaryOperator>
void matrix_for_each(int m, int n, T *p, int ld, TernaryOperator op) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < m; ++i) {
            op(i, j, p[i]);
        }
        p += ld;
    }
}
```

(m+1)n arithmetic op.

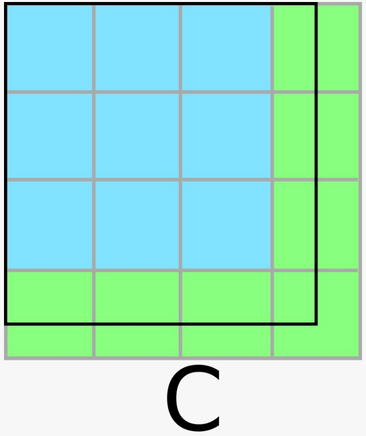
codeplay

Further optimizations

Memory bandwidth no longer an issue.

Focus on decreasing the volume of “useless” arithmetic instructions.

- Address calculation.
- **Bound checking.**
  - Skip bound checking in internal tiles.
  - Bound check in external tiles.



Naive implementation

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$$

AMD R9 Nano

8 Tflop/s peak performance

500 GB/s (125 Gfloat/s) bandwidth

SYCL. ComputeCpp™

Map one work item to each element of c_{ij} and loop over $a_{i:}$ and $b_{:j}$. → ~ 200 Gflop/s

4096-by-4096 matrices

Map one work group per block of C + further optimizations (16-by-16 work group, with 8-by-8 sub-block per work item) → ~ 4 Tflop/s

What next?

- Vectorization: possible performance improvement with vectorized access (vload / vstore).
- Different matrix sizes: If *C* is small, the number of matrix blocks might be too small to utilize the GPU.
  - Use smaller blocks? Less data reuse!
  - Use multiple work groups per block? Race conditions! (need atomic operations)
- Implement other BLAS 3 routines (optimization ideas should be similar)

Takeaway

- Be careful with abstractions.
- Just “throwing in” a lot of computing power into a chip is not enough.
  - Need to strike a balance between computing, memory bandwidth, on-chip memory.

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you! Questions?

@codeplaysoft

info@codeplay.com

codeplay.com