OpenMP 4.0 Accelerator Programming Model & the AClang Compiler

Marcio M Pereira

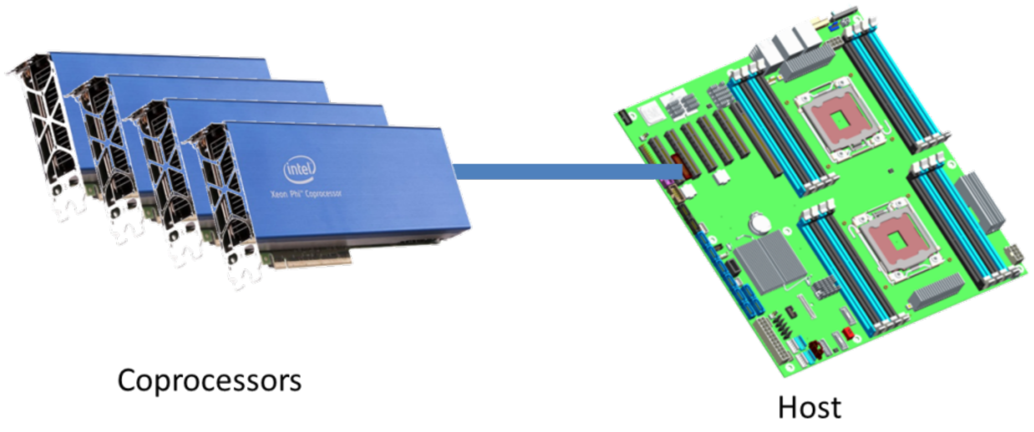
IC/Unicamp

Agenda

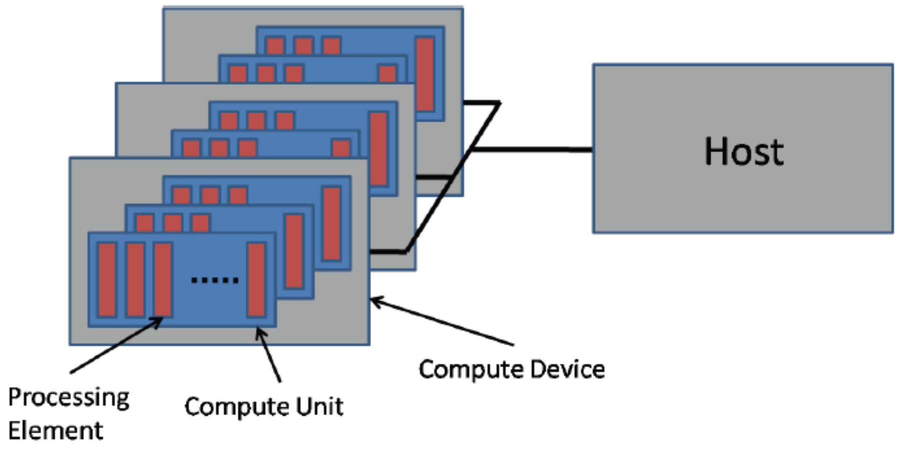
- What is an Accelerator in OpenMP?
- Execution Model and Data Model
- **target** Construct and Accelerator-specific Constructs
- Examples
- The AClang Compiler

Accelerators

- In how differs an accelerator from just another core?
  - ➢different functionality, i.e. optimized for something special
  - ➢different (possibly limited) instruction set
  - → heterogeneous device



Host and Co-processors



Host and GPUs

Heterogeneous device model

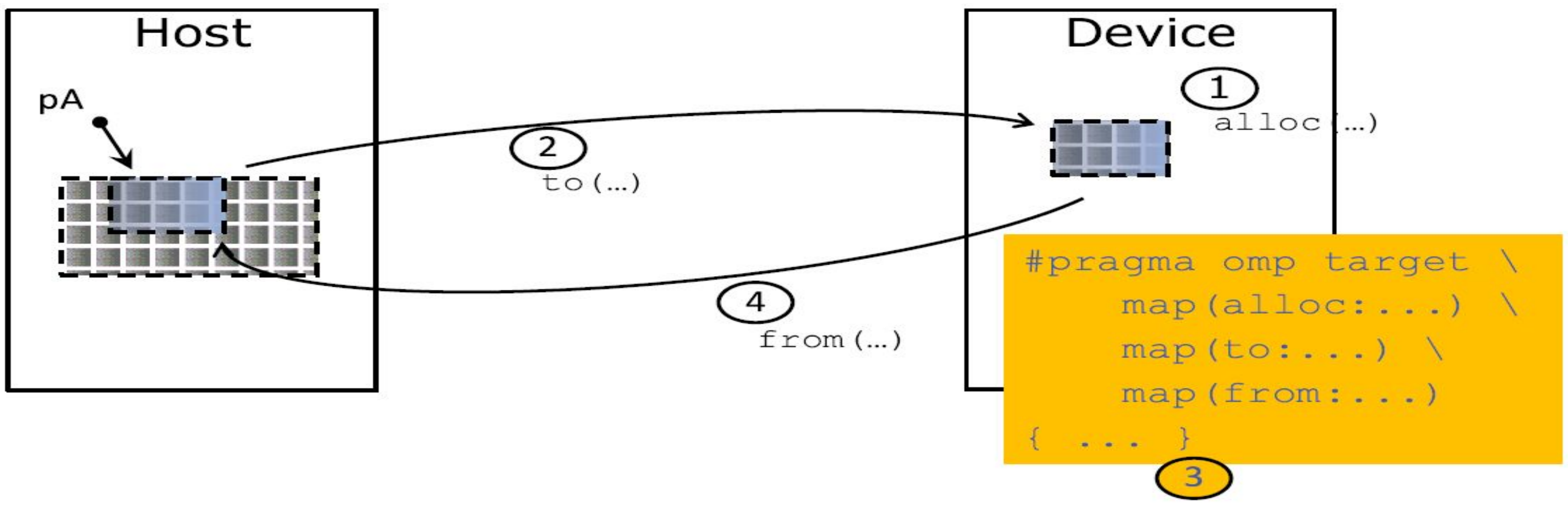
- Assumptions used as design goals for OpenMP 4.0:
  - every accelerator device is attached to one host device
  - it may not be programmable in the same language as the host
  - It may not implement all operations available on the host
  - it may or may not share memory with the host device
  - some accelerators are specialized for loop nests

Execution Model

- Host-centric: the execution of an OpenMP program starts on the host device and it may offload target regions to target devices
- If a target device is not present, or not supported, or not available, the target region is executed by the host device
- If a construct creates a data environment, the data environment is created at the time the construct is encountered

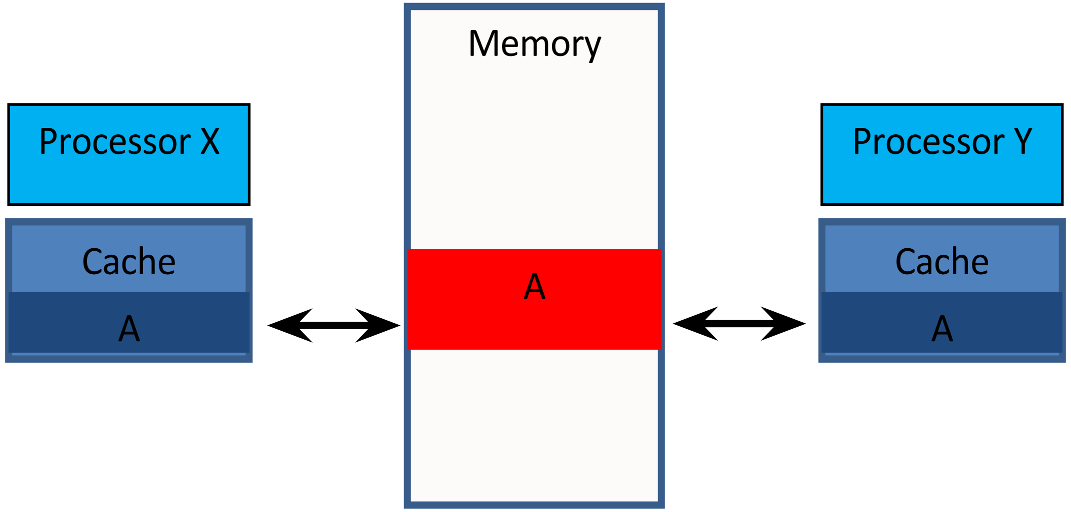
Data environment

- Data environment is lexically scoped
- Data environment is destroyed at closing curly brace
- Allocated buffers/data are automatically released



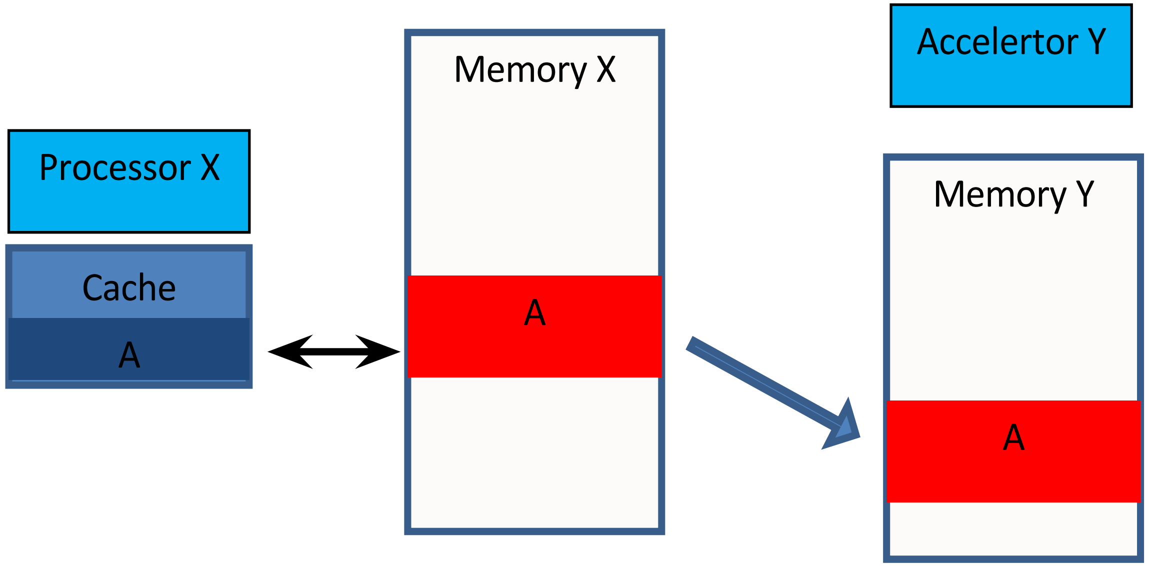
Data mapping: shared or distributed memory

Shared memory



- The corresponding variable in the device data environment *may* share storage with the original variable.
- Writes to the corresponding variable may alter the value of the original variable.

Distributed memory



OpenMP 4.0 device constructs

- the `target` construct transfers the control flow to the target device
  - the map clauses control direction of data flow
  - array notation is used to describe array length
- the `target data` construct creates a scoped device data environment
  - the map clauses control direction of data flow
  - the device data environment is valid through the lifetime of the target data region
- use `target update` to request data transfers from within a target data region

map clause

- Map a variable from the current task's data environment to the device data environment associated with the construct
  - **alloc**-type: each new corresponding list item has an undefined initial value
  - **to**-type: each new corresponding list item is initialized with the original list item's value
  - **from**-type: declares that on exit from the region the corresponding list item's value is assigned to the original list item
  - **tofrom**-type: the default, combination of to and from

target construct example

- Use **target** construct to:
  - ➢Transfer control from the host to the device
  - ➢Establish a device data environment (if not yet done)
- Host thread waits until offloaded region completed

```
void mvt_gpu(float* a, float* x1, float* x2,
                              float* y1, float* y2)
{
  #pragma omp target device(GPU) \
                         map(to: a[:N*N], y1[:N]) \
                         map(tofrom: x1[:N])
  #pragma omp parallel for
  for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
      x1[i] = x1[i] + a[i*N + j] * y1[j];

  #pragma omp target device(GPU) \
                         map(to: a[:N*N], y2[:N]) \
                         map(tofrom: x2[:N])
  {
    #pragma omp parallel for
    for (int i=0; i<N; i++)
      for (int j=0; j<N; j++)
        x2[i] = x2[i] + a[j*N + i] * y2[j];
  }
}
```

data environment example

- Create a data environment to keep data on devices
- Avoid frequent transfers to keep data on devices
- Pre-allocate temporary fields

```
#pragma omp target data device (GPU) \
                        map(alloc: tmp[:N]) \
                        map(to: input[:N]) \
                        map(from: output[:N])
{
  #pragma omp target device (GPU) \
  #pragma omp parallel for
  for (i=0; i<N; i++)
    tmp[i] = some_computation(input[i], i);

  do_some_other_stuff_on_host();

  #pragma omp target device (GPU) \
  #pragma omp parallel for
  for (i=0; i<N; i++)
    output[i] = final_computation(tmp[i], i);
}
```

target update construct example

- Sync input[N] between host and target

```
#pragma omp target data device(GPU) \
                    map(alloc: tmp[:N]) \
                    map(to: input[:N]) \
                    map(from: output[:N])
{
  #pragma omp target device(GPU)
  #pragma omp parallel for
  for (int i=0; i<N; i++)
    tmp[i] = some_computation(input[i], i);

  update_input_array_on_host() ;

  #pragma omp target update device(GPU) to(input[:N])

  #pragma omp target device(GPU)
  #pragma omp parallel for
  for (int i=0; i<N; i++)
    output[i] = final_computation(input[i], tmp[i]);
}
```

teams construct

- Creates a league to threads teams

- The master thread of each team executes the teams region

- A **team** construct must be “perfectly” nested in a **target** construct

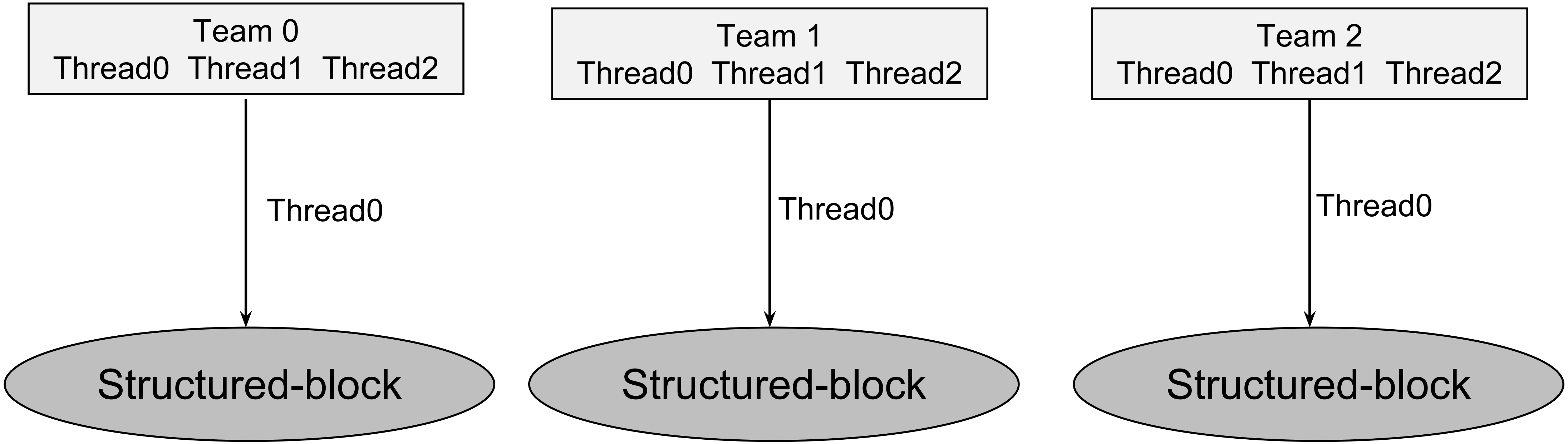
- Only special OpenMP construct can be nested inside a teams contruct:

  - **distribute**
  - **parallel for**

teams execution model

#pragma omp teams num_teams(3), num_threads(3)

structured-block

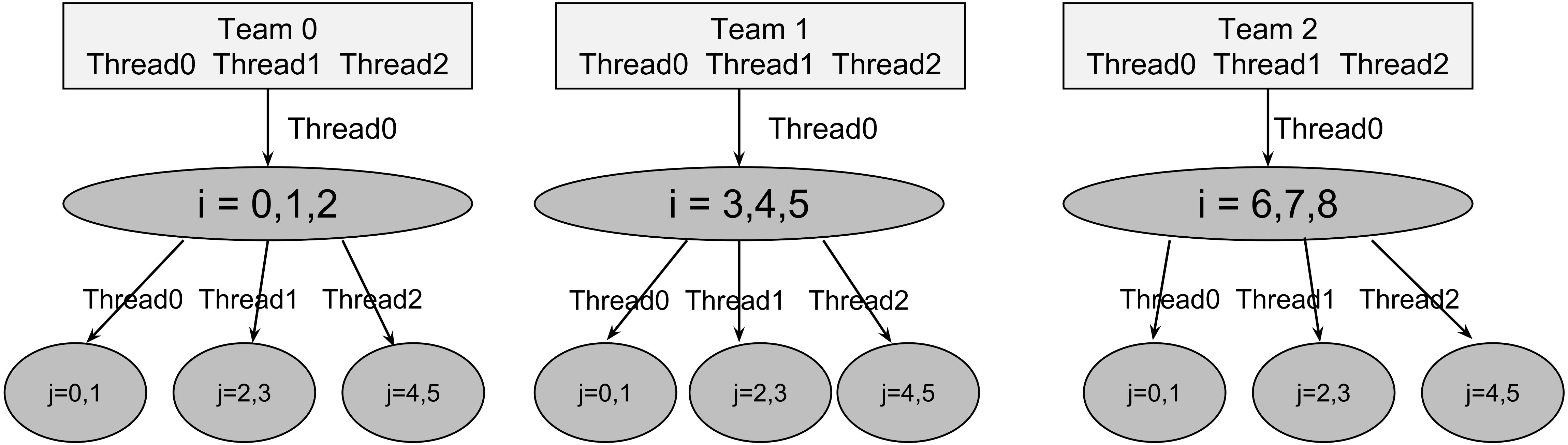


distribute construct

- Iterations distributed among master threads of all teams
- Specify to the loops only
- Must be closed nested to the teams construct
- Workshare among teams to exploit the parallelism on the target device

teams with **distribute** construct

```
#pragma omp teams num_teams(3), num_threads(3)
 #pragma omp distribute
     for (int i=0; i<9; i++) {
       #   pragma omp parallel for
             for (int j=0;j<6; j++)  {
```



saxpy example

```
#pragma omp target data device(GPU) map(to: X[:N])
{
  #pragma omp target map(tofrom: Y[:N])
  #pragma omp teams num_teams(nblocks) \
                    num_threads(nthreads)
  #pragma omp distribute
  for (int i=0; i<N; i+=nblocks)
  {
    #pragma omp parallel for
    for (int j=i; j<i+nblocks; j++)
      Y[j] = a  X[j] + Y[j];
  }
}
```

declare target directive

- Specifies that [static] variables and functions are mapped to a device
- if a list item is a function then a device-specific version of the routines is created that can be called from a target region
- if a list item is a variable then the original variable is mapped to a corresponding variable in the initial device data environment for all devices
- all declarations and definitions for a function must have a declare target directive

```
#pragma omp declare target
  float some_computation(float f, int i) {
    // ... some code ...
  }

  float final_computation(float f, int i) {
    // ... some code ...
  }
#pragma omp end declare target
...
```

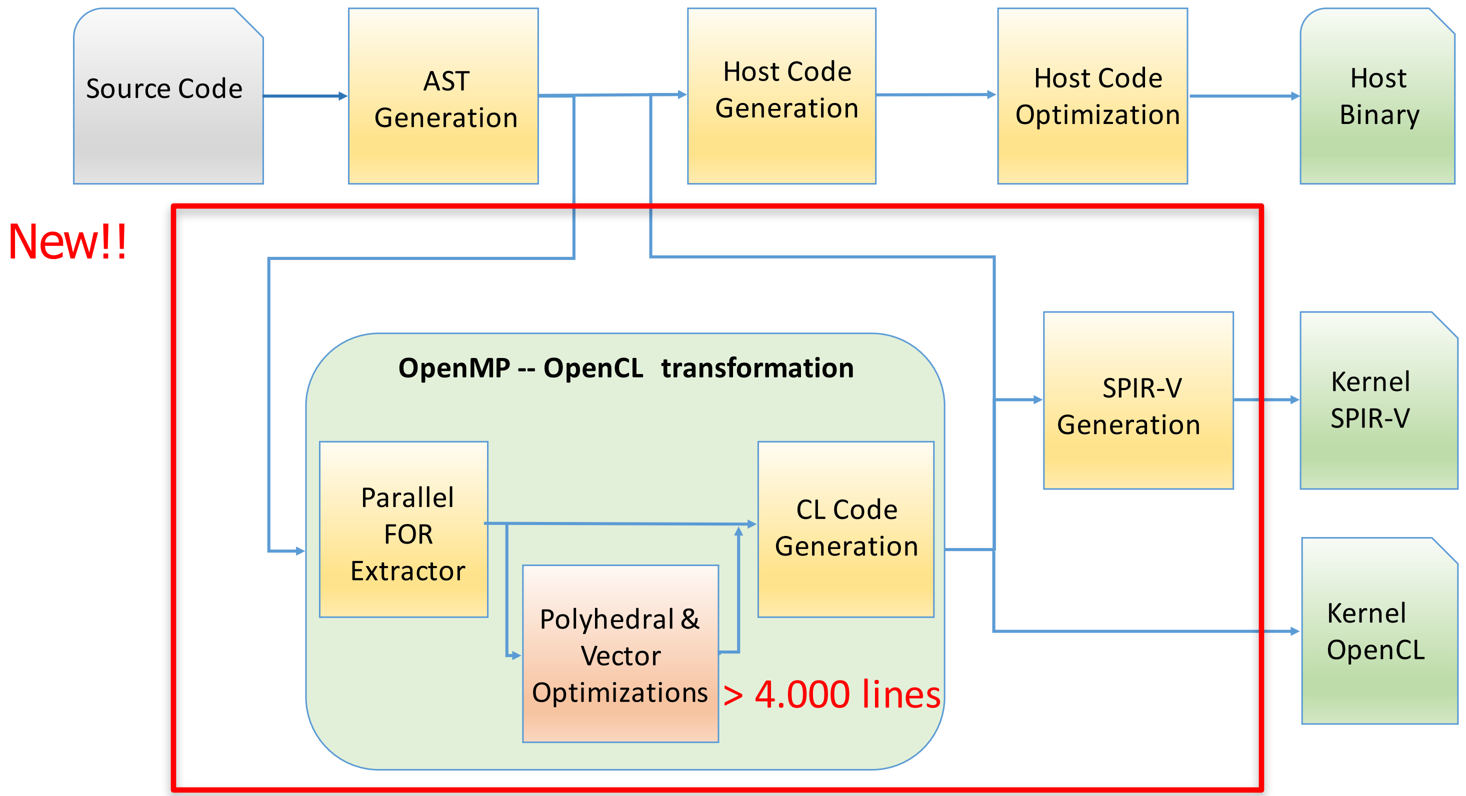
The AClang Compiler

- Started at march/2015
- Main focus: accelerate applications on mobile devices
  - ➢OpenCL is the language to dispatch kernels on GPUs
- No support of OpenMP 4.0 on llvm/clang trunk
- OpenMP group on llvm with focus on NVIDIA/Cuda
- Solution
  - ➢Map (part of) OpenMP 4.0 to OpenCL
  - ➢ but not source-to-source

Deliverable: AClang framework

Very advanced optimizing compiler

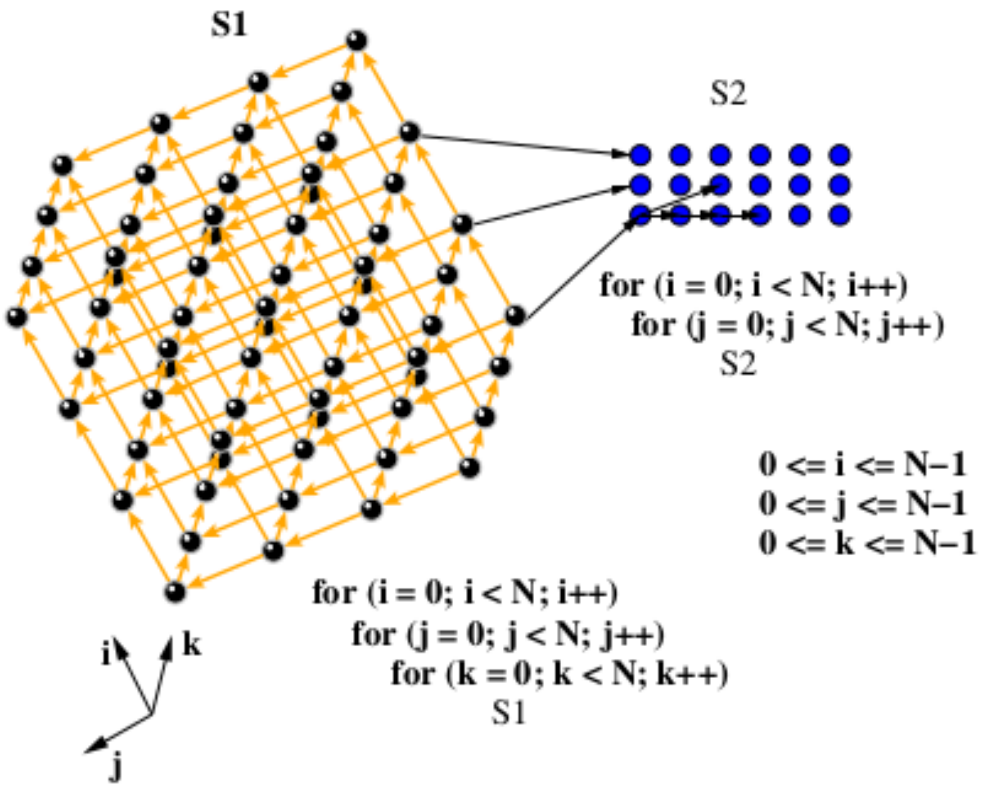
- > 6.000 lines just for the OpenMP – OpenCL translation
- > 2.000 lines for the GPU runtime library

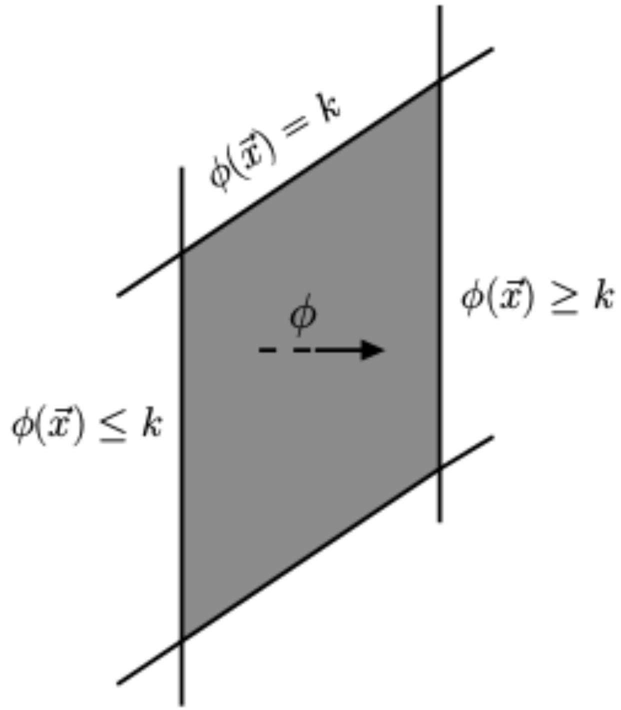


Deliverable: AClang optimizations

Use of polyhedral techniques for OpenCL kernels to do:

- loop tiling,
- vectorization





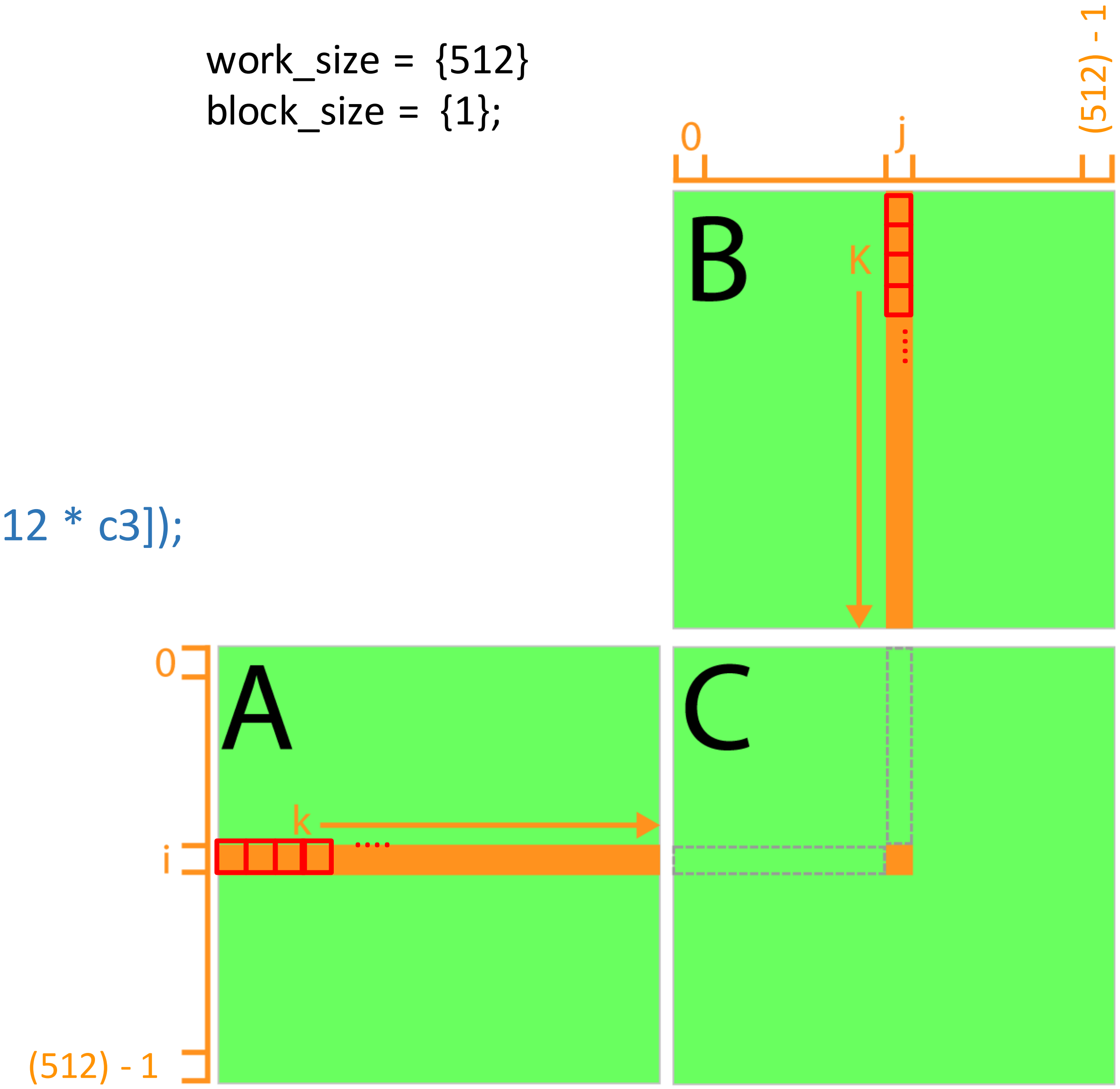
Case Study: Matrix multiplication

```
void matrix_multiply (float *A, float *B, float *C) {

  for (int i = 0; i < 512; i++) {
    for (int j = 0; j < 512; j++) {
      for (int k = 0; k < 512; ++k) {
        C[i * 512 + j] += A[i * 512 + k] * B[k * 512 + j];
      }
    }
  }
}
```

Extracting Kernel : no-optimization

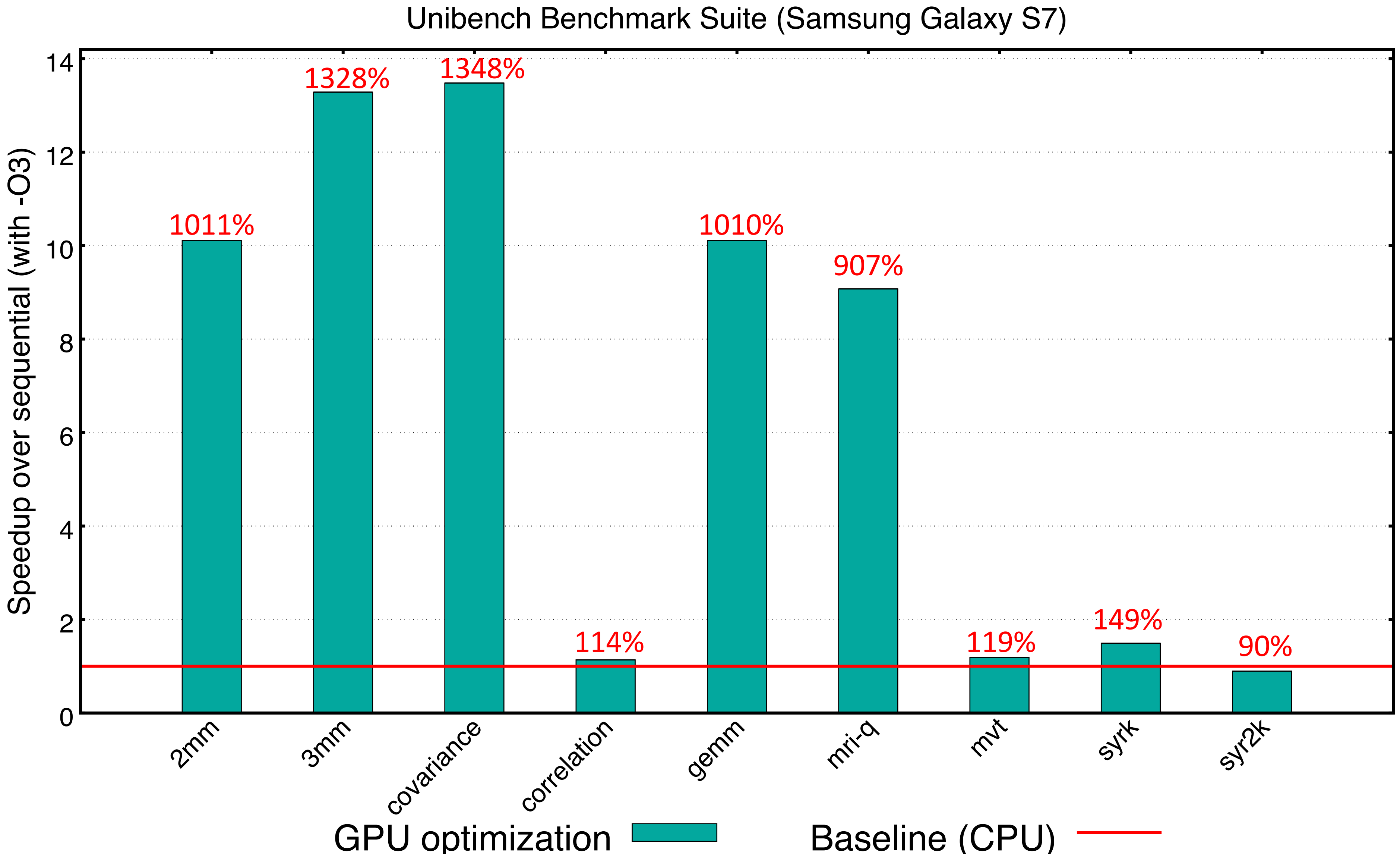
```
__kernel void kernel_mm_1 (__global float *A,
                           __global float *B,
                           __global float *C) {
  int b0 = get_group_id (0);
  int t0 = get_local_id (0);

  for (int c2 = 0; c2 <= 511; c2 += 1)
    for(int c3 = 0; c3 <= 511; c3 += 1)
      C[512 * b0 + c2] += (A[512 * b0 + c3] * B[c2 + 512 * c3]);
  }
}
```

work_size = {512}
block_size = {1};



"kernel is just similar to C function"

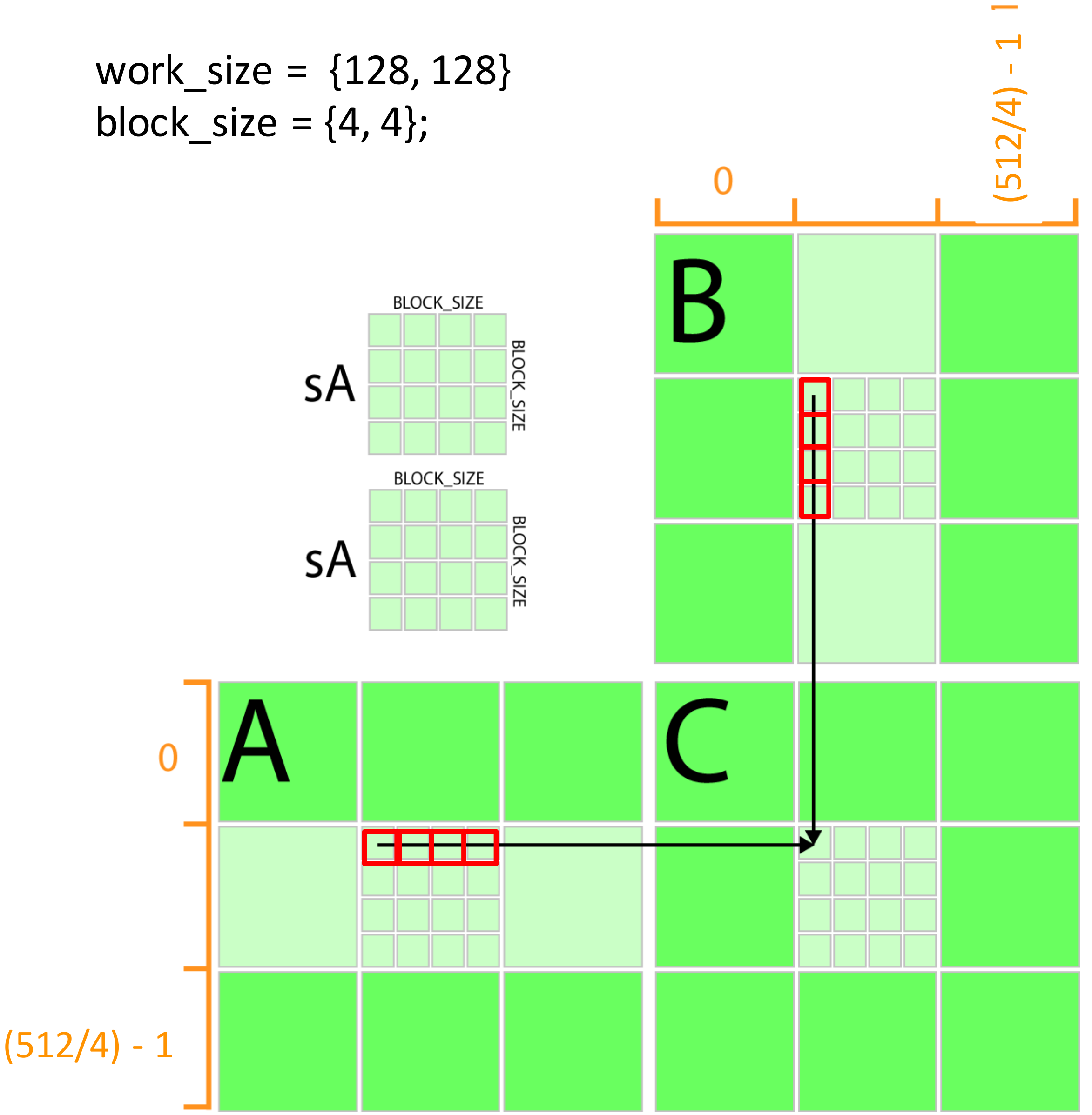
GPU Optimization



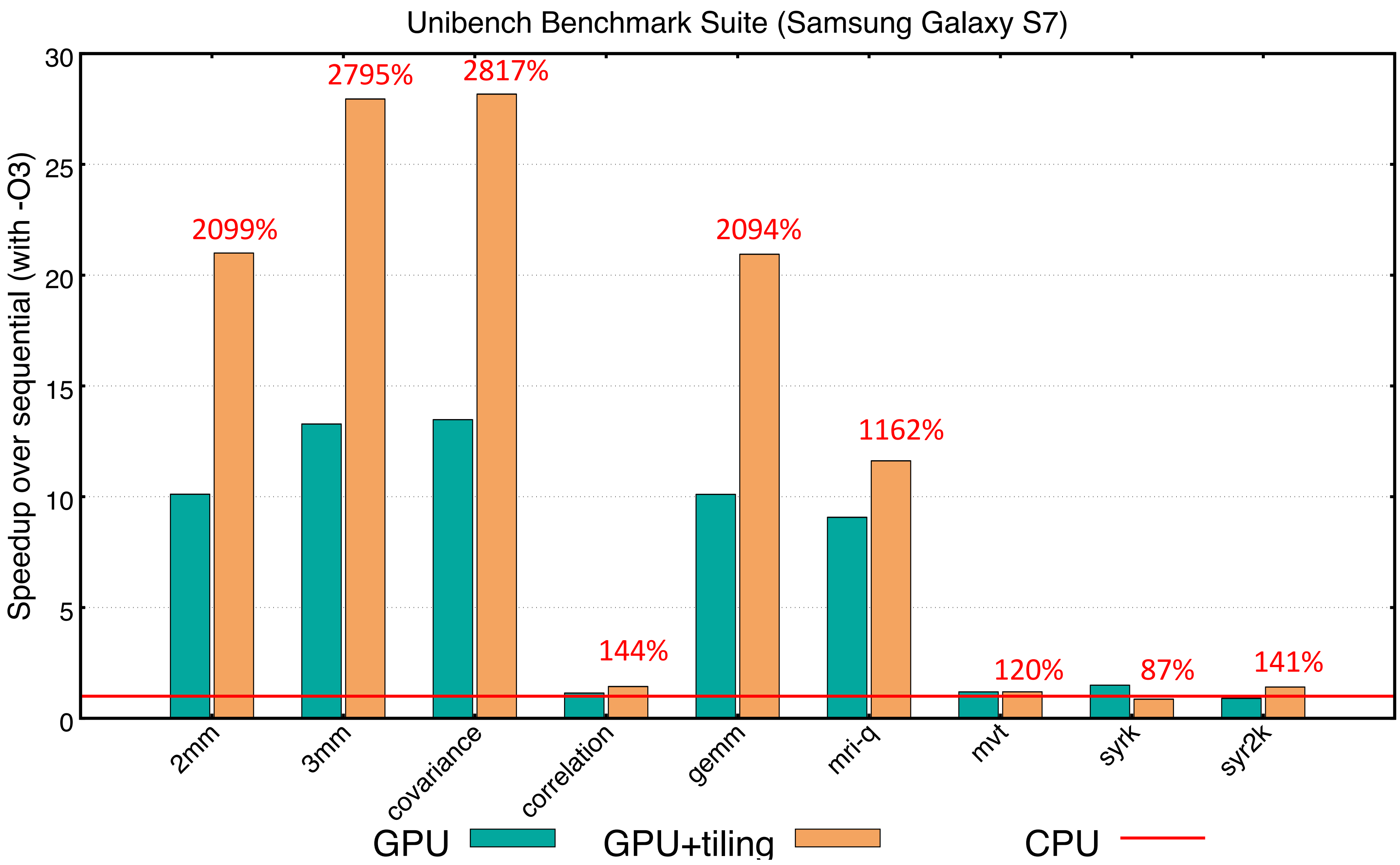
Applying tile optimization (tile-size=4)

```
__kernel void kernel_mm_2 (__global float *A,
                           __global float *B,
                           __global float *C) {

   int b0 = get_group_id(0), b1 = get_group_id(1);
   int  t0 = get_local_id(0),    t1 = get_local_id(1);

   for (int c2 = 0; c2 < 512; c2 += 4) {
      for (int c5 = 0; c5 <= 3; c5 += 1)
         C[4 * (512 * b0 + b1) + 512 * t0 + t1] +=
            (A[512 * ( 4*b0 + t0 ) + (c2 + c5)]  *
             B[4*b1 + t1 + 512 * (c2 + c5)]);

   }
}
```

work_size = {128, 128}
block_size = {4, 4};



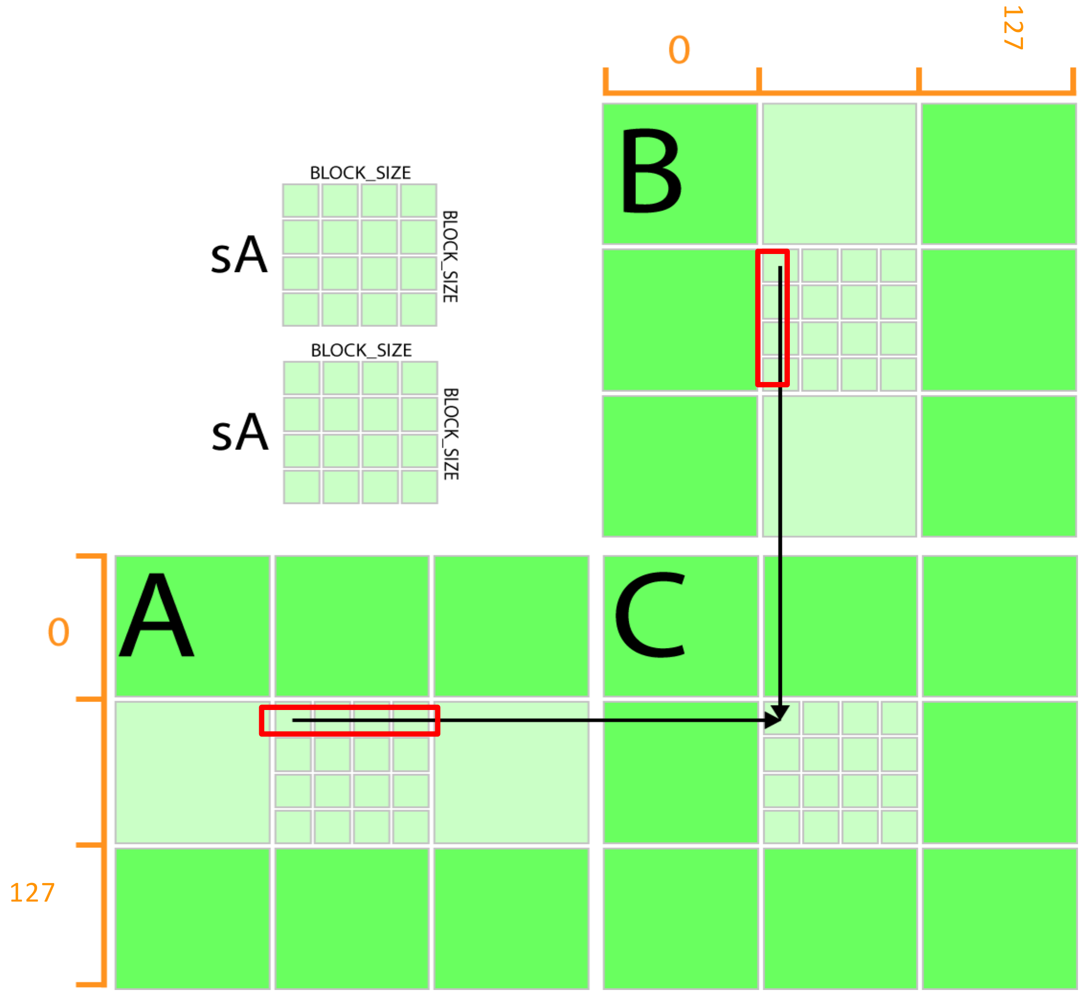
GPU + tile optimization Results



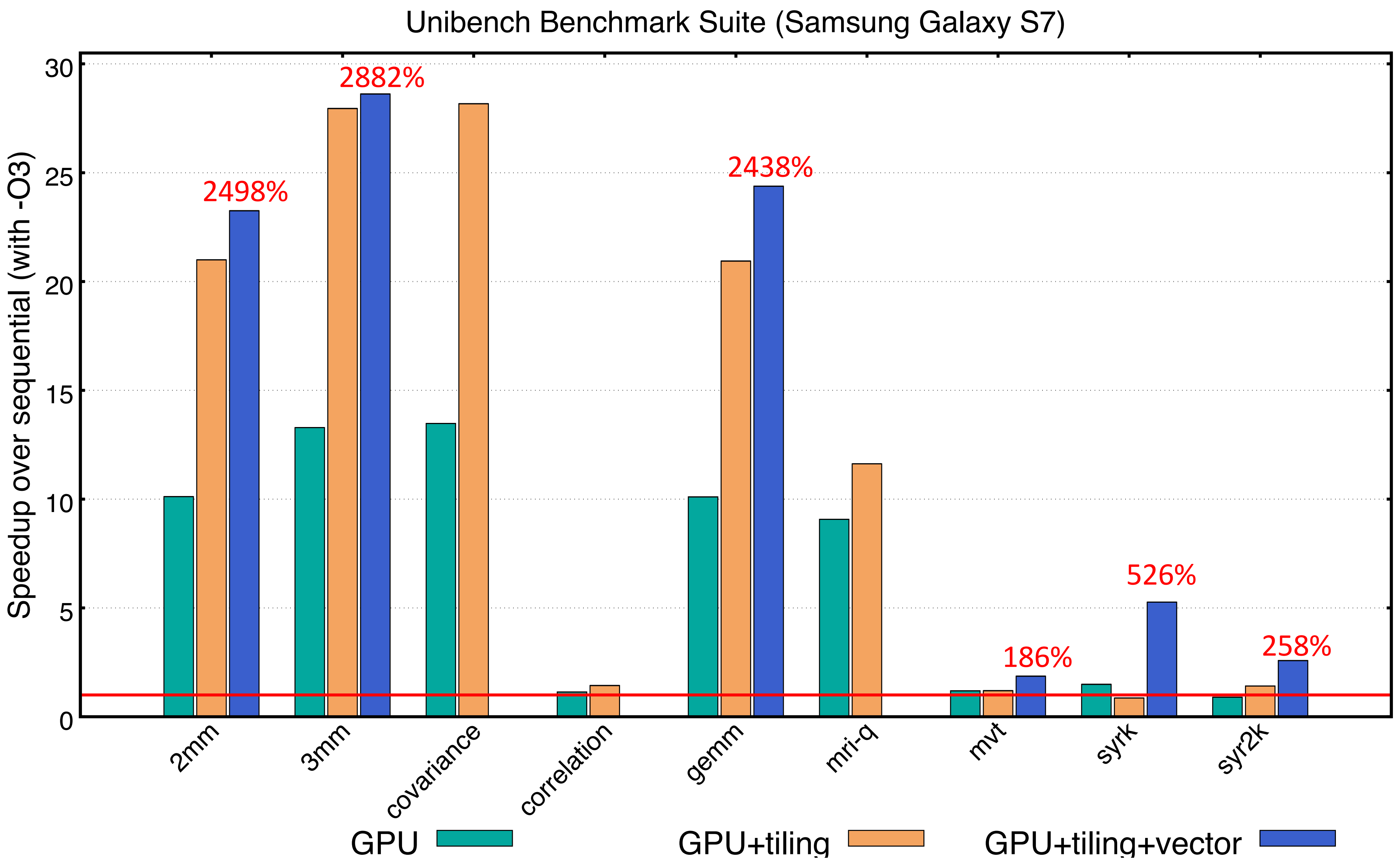
Applying vector optimization

```
__kernel void kernel_mm_3 (__global float *A,
                           __global float *B,
                           __global float *C) {
    int b0 = get_group_id(0), b1 = get_group_id(1);
    int  t0 = get_local_id(0),    t1 = get_local_id(1);
    __private float4 _ft0  = {0., 0., 0., 0.};
    __private float4 _ft1, _ft2;
    for (int c2 = 0; c2 < 512; c2 += 4) {
      _ft1 = vload4 (0, &A[512 * (4 * b0 + t0) + c2]);
      _ft2.x  = B[4 * b1 + t1 + 512 * c2];
      _ft2.y  = B[4 * b1 + t1 + 512 * (c2 + 1)];
      _ft2.z  = B[4 * b1 + t1 + 512 * (c2 + 2)];
      _ft2.w = B[4 * b1 + t1 + 512 * (c2 + 3)];
      _ft0   += _ft1 * _ft2;
    }
    C[4 * (512 * b0 + b1) + 512 * t0 + t1] =
        _ft0.x + _ft0.y + _ft0.z + _ft0.w;
}
```

work_size = {128,128}
block_size = {4, 4};



GPU +vector optimization Results



How to use the AClang Compiler?

Some examples

- clang -fopenmp -omptargets=opencl-unknown-unknown -rtl-mode=verbose -o test test.c
- clang –fopenmp=lgomp -omptargets=opencl-unknown-unknown -opt-poly=vectorize -o test test.c
- clang –fopenmp=liomp5 -omptargets=spir64-unknown-unknown -opt-poly=tile -tile-size=32 -o test test.c

Flavors

rtl-mode: none (default), profile, verbose, all

opt-poly: node (default), tile, vectorize

tile-size: default=16

Questions?