Review Exercise 4

Question 1: Amdahl's Law

- No problems at all
- Reasoning whether to buy more nodes was very good!
- Expect something like this in the exam

Review Exercise 4

Question 2: Manual vectorization of reduction

Kernel analysis:

$$r = \sum_{i=1}^{N} a_i$$

```
1 template <typename T>
2 static inline T gold_red(const T* const ary, const size_t N)
3 {
4     T sum = 0.0;
5     for (size_t i = 0; i < N; ++i)
6         sum += ary[i];
7     return sum;
8 }
```

OI:

$$I = \frac{1}{\text{sizeof(T)}}$$

Expectation: Memory Bound!

Review Exercise 4

Question 2: Manual vectorization of reduction

Vectorization:

```
1 template <typename T>
2 static inline T gold_red(const T* const ary, const size_t N)
3 {
4     T sum = 0.0;
5     for (size_t i = 0; i < N; ++i)
6         sum += ary[i];
7     return sum;
8 }
```

Review Exercise 4

Question 2: Manual vectorization of reduction

Vectorization:

```
1 template <typename T>
2 static inline T gold_red(const T* const ary, const size_t N)
3 {
4     T sum = 0.0;
5     for (size_t i = 0; i < N; ++i)
6         sum += ary[i];
7     return sum;
8 }
```

1. Initialize a vector register to zero and use it as a container for the progressing summation.

```
__m128 sum4 = _mm_set1_ps(0.0f);
__m128 sum4 = _mm_setzero_ps();
```

Any of them are OK

Review Exercise 4

Question 2: Manual vectorization of reduction

Vectorization:

```
1 template <typename T>
2 static inline T gold_red(const T* const ary, const size_t N)
3 {
4     T sum = 0.0;
5     for (size_t i = 0; i < N; ++i)
6         sum += ary[i];
7     return sum;
8 }
```

1. Initialize a vector register to zero and use it as a container for the progressing summation.

```
__m128 sum4 = _mm_set1_ps(0.0f);
__m128 sum4 = _mm_setzero_ps();
```

Any of them are OK

2. There are two operations required in the loop body: Load from address and addition, need two intrinsics for this. Because we work with vector registers, the loop counter must have the correct stride:

```
for (size_t i = 0; i < N; i += simd_width)
    sum4 = _mm_add_ps(sum4, _mm_load_ps(ary+i));
```

This is all you need

Review Exercise 4

Question 2: Manual vectorization of reduction

Vectorization:

```
1 template <typename T>
2 static inline T gold_red(const T* const ary, const size_t N)
3 {
4     T sum = 0.0;
5     for (size_t i = 0; i < N; ++i)
6         sum += ary[i];
7     return sum;
8 }
```

1. Initialize a vector register to zero and use it as a container for the progressing summation.

```
__m128 sum4 = _mm_set1_ps(0.0f);
__m128 sum4 = _mm_setzero_ps();
```

Any of them are OK

2. There are two operations required in the loop body: Load from address and addition, need two intrinsics for this. Because we work with vector registers, the loop counter must have the correct stride:

```
for (size_t i = 0; i < N; i += simd_width)
    sum4 = _mm_add_ps(sum4, _mm_load_ps(ary+i));
```

This is all you need

3. The function returns a scalar. The vector register `sum` must also be reduced at the end:

```
float last[4];
_mm_storeu_ps(last, sum4);
return last[0] + last[1] + last[2] + last[3];
```

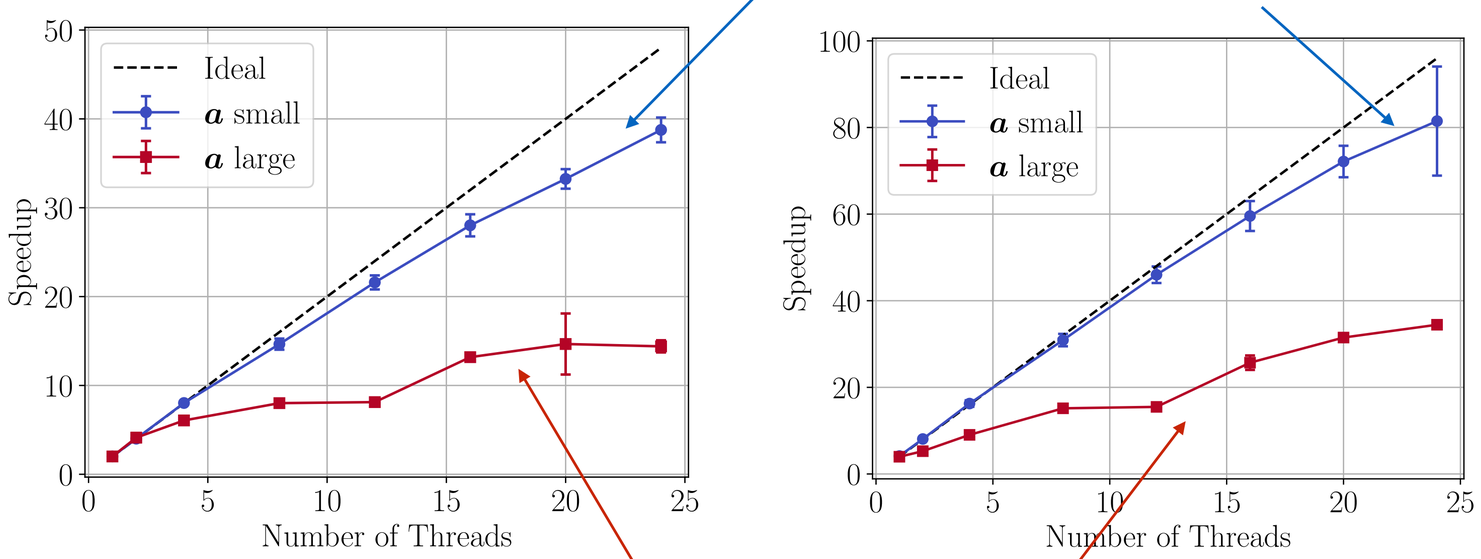
Note: Some allocated the `last` *variable using an aligned memory allocator. The OS call is much more expensive than just using one* `_mm_storeu_ps` *instruction. Others used* `_mm_hadd_ps` *which is SSE3 — we targeted SSE/SSE2 :-)*

Review Exercise 4

Question 2: Manual vectorization of reduction

Performance:

Exploit temporal locality (caches)

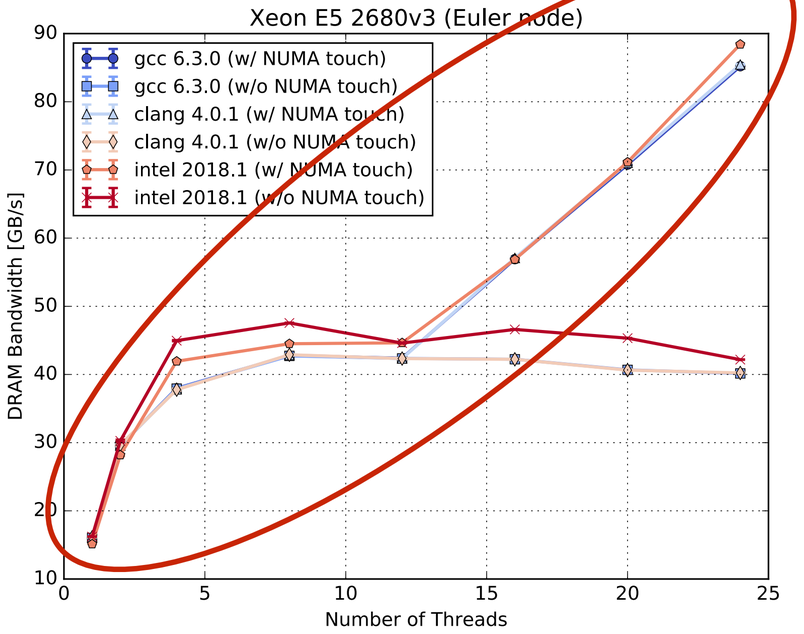


(a) 2-way SIMD (b) 4-way SIMD

Figure 1: Xeon E5-2680v3 node on Euler (GCC 6.3.0)

Can not exploit temporal locality for reduction kernel in real life

STREAM Benchmark:



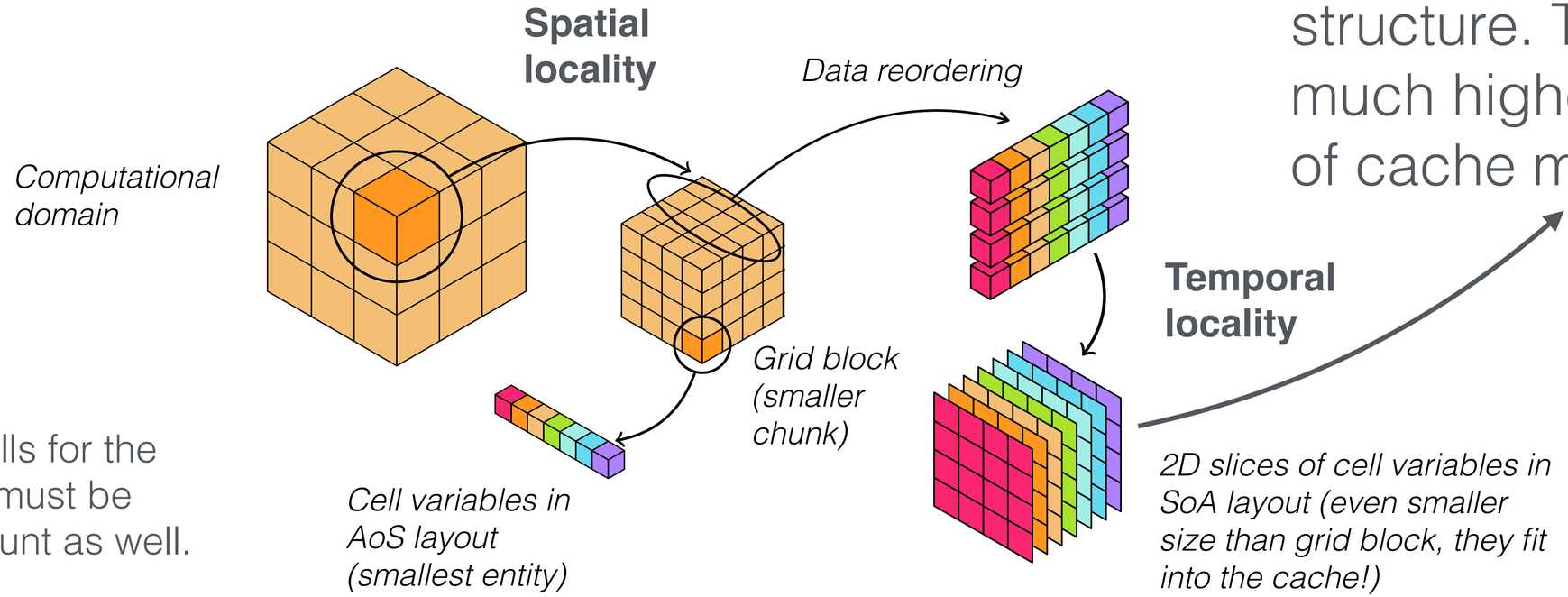
Memory bound as we would expect from kernel analysis

Review Exercise 4

Question 2: Manual vectorization of reduction

Can not exploit cache memories efficiently with a reduction. What about other kernels?

Consider 3D stencil codes:



Stencil code is applied to the slice data structure. This allows much higher utilization of cache memories!

Note: Ghost cells for the stencil kernels must be taken into account as well.

Review Exercise 4

Question 3: ISPC

Baseline (scalar code):

Before we optimize code with vector intrinsics, it is important to have a baseline code that performs as best as it can.

```
template <typename T>
static void gemm_bad_access_pattern(const T* const A, const T* const B, T* const C,
        const int p, const int r, const int q)
{
    for (int i = 0; i < p; ++i)
        for (int j = 0; j < q; ++j)
        {
            T sum = 0.0;
            for (int k = 0; k < r; ++k)
                sum += A[i*r + k] * B[k*q + j];
            C[i*q + j] = sum;
        }
}
```

Review Exercise 4

Question 3: ISPC

Baseline (scalar code):

Before we optimize code with vector intrinsics, it is important to have a baseline code that performs as best as it can.

```
template <typename T>
static void gemm_bad_access_pattern(const T* const A, const T* const B, T* const C,
        const int p, const int r, const int q)
{
    for (int i = 0; i < p; ++i)
        for (int j = 0; j < q; ++j)
        {
            T sum = 0.0;
            for (int k = 0; k < r; ++k)
                sum += A[i*r + k] * B[k*q + j];
            C[i*q + j] = sum;
        }
}
```

One inner product

Bad because of non-unit stride access into matrix B

Partial workaround: Switch the two innermost loops. Still bad because of many memory references into matrix C

Review Exercise 4

Question 3: ISPC

Baseline (scalar code):

Before we optimize code with vector intrinsics, it is important to have a baseline code that performs as best as it can.

```
template <typename T>
static void gemm_bad_access_pattern(const T* const A, const T* const B, T* const C,
        const int p, const int r, const int q)
{
    for (int i = 0; i < p; ++i)
        for (int j = 0; j < q; ++j)
        {
            T sum = 0.0;
            for (int k = 0; k < r; ++k)
                sum += A[i*r + k] * B[k*q + j];
            C[i*q + j] = sum;
        }
}
```

One inner product

Bad because of non-unit stride access into matrix B

Partial workaround: Switch the two innermost loops. Still bad because of many memory references into matrix C

Use temporary work buffer that fits into cache:

```
template <typename T>
static void gemm_good_access_pattern(const T* const A, const T* const B,
        T* const C, const int p, const int r, const int q)
{
    T tile[_VTILE_][_HTILE_];

    for (int i = 0; i < p; i += _VTILE_)
        for (int j = 0; j < q; j += _HTILE_)
        {
            for (int tv = 0; tv < _VTILE_; ++tv)
                for (int th = 0; th < _HTILE_; ++th)
                    tile[tv][th] = (T)0.0;

            for (int k = 0; k < r; ++k)
                for (int tv = 0; tv < _VTILE_; ++tv)
                {
                    const T Aik = A[(i+tv)*r + k];
                    for (int th = 0; th < _HTILE_; ++th)
                        tile[tv][th] += Aik * B[k*q + j + th];
                }

            for (int tv = 0; tv < _VTILE_; ++tv)
                for (int th = 0; th < _HTILE_; ++th)
                    C[(i+tv)*q + j + th] = tile[tv][th];
        }
}
```

_VTILE_*_HTILE_ inner products

Review Exercise 4

Question 3: ISPC

Baseline (scalar code):

Before we optimize code with vector intrinsics, it is important to have a baseline code that performs as best as it can.

```
template <typename T>
static void gemm_bad_access_pattern(const T* const A, const T* const B, T* const C,
        const int p, const int r, const int q)
{
    for (int i = 0; i < p; ++i)
        for (int j = 0; j < q; ++j)
        {
            T sum = 0.0;
            for (int k = 0; k < r; ++k)
                sum += A[i*r + k] * B[k*q + j];
            C[i*q + j] = sum;
        }
}
```

One inner product

Bad because of non-unit stride access into matrix B

Partial workaround: Switch the two innermost loops. Still bad because of many memory references into matrix C

Use temporary work buffer that fits into cache:

```
template <typename T>
static void gemm_good_access_pattern(const T* const A, const T* const B,
        T* const C, const int p, const int r, const int q)
{
    T tile[_VTILE_][_HTILE_];

    for (int i = 0; i < p; i += _VTILE_)
        for (int j = 0; j < q; j += _HTILE_)
        {
            for (int tv = 0; tv < _VTILE_; ++tv)
                for (int th = 0; th < _HTILE_; ++th)
                    tile[tv][th] = (T)0.0;

            for (int k = 0; k < r; ++k)
                for (int tv = 0; tv < _VTILE_; ++tv)
                {
                    const T Aik = A[(i+tv)*r + k];
                    for (int th = 0; th < _HTILE_; ++th)
                        tile[tv][th] += Aik * B[k*q + j + th];
                }

            for (int tv = 0; tv < _VTILE_; ++tv)
                for (int th = 0; th < _HTILE_; ++th)
                    C[(i+tv)*q + j + th] = tile[tv][th];
        }
}
```

_VTILE_*_HTILE_ inner products

Initialize

Inner product

Store

Good accessing pattern — 10x faster!

Review Exercise 4

Question 3: ISPC

ISPC implementation:

Using a temporary work buffer makes life much easier!

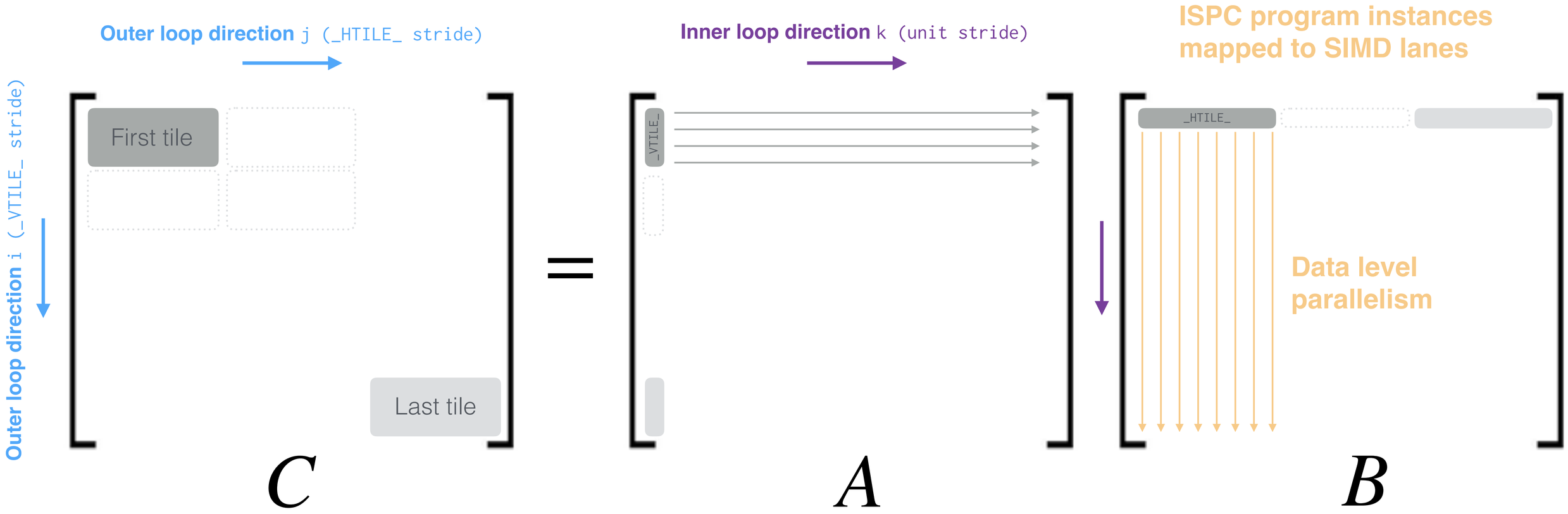
Recall:
Memory references are fetched from linear memory. Of course, this also applies when loading data into vector registers. Not all problems you are going to work with will play nice with this issue! Reduction of Q2 was nice, GEMM here is not! We need to make it nice… Unfortunately, this is not trivial in general.

Review Exercise 4

Question 3: ISPC

ISPC implementation:

Vectorization strategy: How should we map ISPC program instances to our problem?



Review Exercise 4

Question 3: ISPC

ISPC implementation:

The very same structure as for the scalar code, except that _HTILE_ dimension is mapped to SIMD lanes.

```
export void gemm_ISPC(
        const uniform Real* uniform const A,
        const uniform Real* uniform const B,
        uniform Real* uniform const C,
        const uniform int p, const uniform int r, const uniform int q)
{
    uniform Real tile[_VTILE_][_HTILE_];
    uniform Real Aik[_VTILE_];

    for (uniform int i = 0; i < p; i += _VTILE_)
        for (uniform int j = 0; j < q; j += _HTILE_)
        {
            for (uniform int tv = 0; tv < _VTILE_; ++tv)
                foreach (th = 0 ... _HTILE_)
                    tile[tv][th] = (Real)0.0;

            for (uniform int k = 0; k < r; ++k)
            {
                for (uniform int tv = 0; tv < _VTILE_; ++tv)
                    Aik[tv] = A[(i+tv)*r + k];

                for (uniform int tv = 0; tv < _VTILE_; ++tv)
                    foreach (th = 0 ... _HTILE_)
                        tile[tv][th] += Aik[tv] * B[k*q + j + th];
            }

            for (uniform int tv = 0; tv < _VTILE_; ++tv)
                foreach (th = 0 ... _HTILE_)
                    C[(i+tv)*q + j + th] = tile[tv][th];
        }
}
```

Review Exercise 4

Question 3: ISPC

ISPC implementation:

The very same structure as for the scalar code, except that _HTILE_ dimension is mapped to SIMD lanes.

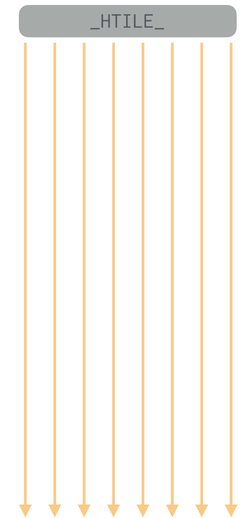
Note: If `export` is missing, ISPC will not generate the kernel declaration in the generated header files.

Note: _HTILE_ could be any size, ideally you want it to be an integer multiple of the ISPC gang size because that is how ISPC maps program instances to SIMD lanes.

```
1  export void gemm_ISPC(
2          const uniform Real* uniform const A,
3          const uniform Real* uniform const B,
4          uniform Real* uniform const C,
5          const uniform int p, const uniform int r, const uniform int q)
6  {
7      uniform Real tile[_VTILE_][_HTILE_];
8      uniform Real Aik[_VTILE_];
9
10     for (uniform int i = 0; i < p; i += _VTILE_)
11         for (uniform int j = 0; j < q; j += _HTILE_)
12         {
13             for (uniform int tv = 0; tv < _VTILE_; ++tv)
14                 foreach (th = 0 ... _HTILE_)
15                     tile[tv][th] = (Real)0.0;
16
17             for (uniform int k = 0; k < r; ++k)
18             {
19                 for (uniform int tv = 0; tv < _VTILE_; ++tv)
20                     Aik[tv] = A[(i+tv)*r + k];
21
22                 for (uniform int tv = 0; tv < _VTILE_; ++tv)
23                     foreach (th = 0 ... _HTILE_)
24                         tile[tv][th] += Aik[tv] * B[k*q + j + th];
25             }
26
27             for (uniform int tv = 0; tv < _VTILE_; ++tv)
28                 foreach (th = 0 ... _HTILE_)
29                     C[(i+tv)*q + j + th] = tile[tv][th];
30         }
31 }
```

Outer loops

Inner loop



Review Exercise 4

Summary:

- **Question 1:**
  - No general issues. Nice!
- **Question 2:**
  - Some used `_mm_hadd_ps` for horizontal reduction — SSE3 feature and typically slow.
  - Dynamic memory allocation inside performance critical code should be avoided!
  - As the OpenMP benchmark passes a `nthreads` parameter, you should use it when spawning parallel regions — `#pragma omp parallel num_threads(nthreads)`
  - It is wrong to parallelize the loop that collects the sample measurements in the benchmark. Each thread must collect the same number of samples.
- **Question 3:**
  - The baseline kernel has often been implemented naively — The memory access pattern matters a lot for this kernel. The vectorization may yield higher than expected speedups because you compare against a poor baseline.
  - It is important to think about the way you want to access the memory when vectorizing/ISPCing your code. ISPC can still deal with a poor access pattern (gathers) at the cost of performance. For the GEMM kernel we can reorder the data to eliminate this problem and access the data uniformly.

Review Exercise 4

Summary:

- **Question 1:**
  - No general issues. Nice!
- **Question 2:**
  - Some used `_mm_hadd_ps` for horizontal reduction — SSE3 feature and typically slow.
  - Dynamic memory allocation inside performance critical code should be avoided!
  - As the OpenMP benchmark passes a `nthreads` parameter, you should use it when spawning parallel regions — `#pragma omp parallel num_threads(nthreads)`
  - It is wrong to parallelize the loop that collects the sample measurements in the benchmark. Each thread must collect the same number of samples.
- **Question 3:**
  - The baseline kernel has often been implemented naively — The memory access pattern matters a lot for this kernel. The vectorization may yield higher than expected speedups because you compare against a poor baseline.
  - It is important to think about the way you want to access the memory when vectorizing/ISPCing your code. ISPC can still deal with a poor access pattern (gathers) at the cost of performance. For the GEMM kernel we can reorder the data to eliminate this problem and access the data uniformly.

ISPC Makefile:

Intel 64bit ISA

```
1  ISPC ?= ispc
2  ISPCFLAGS ?= --arch=x86-64
3
4  gemm_sse2.o: gemm.ispc
5    $(ISPC) $(ISPCFLAGS) -D_ISPC_SSE2_ \
6      --target=sse2-i32x8 \
7      -o gemm_sse2.o -h gemm_sse2.h $<
8
9  gemm_avx2.o: gemm.ispc
10   $(ISPC) $(ISPCFLAGS) -D_ISPC_AVX2_ \
11     --target=avx2-i32x16 \
12     -o gemm_avx2.o -h gemm_avx2.h $<
```

ISA Extensions

`i32x8`

Addressing mode (integer arithmetic, i32 means 32bit addressing, more efficient than 64bit which is also supported by ISPC)

Gang size (usually 2-4x the SIMD width)