HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 20

Fabian Wermelinger
Harvard University
CS205

Thursday, April 20th 2023

LAST TIME

- Data alignment and how to align memory in code.
- Data layouts concerned with SIMD (data-level parallelism)
- Single-Instruction-Multiple-Data (SIMD):
  - Compiler auto-vectorization.
  - SIMD support in OpenMP
  - Intel intrinsics (x86-64 specific).
  - Manual vectorization of code (SIMD instructions).

TODAY

Main topic: ***SPMD-to-SIMD programming model, more SIMD***

Details:

- Quiz 4
- Explicit versus implicit vectorization (not the same as auto-vectorization)
- The Intel *Implicit SPMD Program Compiler (ISPC)*
- ISPC execution model (reading assignment) → *very similar to GPUs*
- How is "SPMD-to-SIMD" mapped to hardware (execution mask in ISPC)
- Basic C/C++ language extensions in ISPC
  Examples with reductions and GEMM (with performance comparison to Eigen)

ISSUES WITH EXPLICIT VECTORIZATION

- Explicit (manual) vectorization is *laborious* (especially for higher dimensional problems).
- Switching between different ISA extensions (e.g. SSE or AVX) is not straight forward → limited portability *(need two kernels for SSE and AVX)*.
- No direct support of selecting a best suitable kernel depending on the architecture you are running on.
- Development process does not scale.

IMPLICIT VECTORIZATION

- It would be desired to write SIMD code from the point of view of a *single SIMD lane* instead of n SIMD lanes simultaneously (key idea in CUDA).
- Indexing and treatment of multiple SIMD lanes should be done *implicitly*.
- **S**ingle **P**rogram **M**ultiple **D**ata (SPMD) programming model (lecture 11).

INTEL IMPLICIT SPMD PROGRAM COMPILER (ISPC)

- Open source "SPMD-on-SIMD" compiler (https://github.com/ispc/ispc/)
- Documentation: https://ispc.github.io/ispc.html
- Implicit vectorization programming model that exploits DLP using SIMD.
- *Does not make the same assumptions* as a compiler must do when *auto-vectorizing sequential* C/C++ code.
- The SPMD programming model exposes parallelism by mapping individual *SIMD lanes* onto the *hardware*, every source line in the SPMD program is assumed *vectorizable* unless indicated otherwise.
- The SPMD programming model is also used for GPU architectures. The Nvidia CUDA programming model is based on SPMD where *kernel programs are mapped to GPU threads* instead of SIMD lanes. Nvidia calls this architecture SIMT.
- To benefit from ISPC, your data layout must be data oriented (SIMD-friendly → structure of arrays)
- ISPC is based on C with few additional keywords and C++ extensions.

ISPC EXECUTION MODEL

- ISPC is employed via functions → we call them (compute) kernels.
- A kernel is called from C/C++: upon kernel entry the execution is switched from the sequential C/C++ caller model to the SPMD model in the ISPC kernel.
- A number of *program instances* start running *concurrently on the same hardware thread and context*. The group of *running program instances* is called a *gang*.
- ISPC *will not* create threads or implicitly switch contexts of running threads. (ISPC supports creating asynchronous threads. We will not discuss this functionality here because we can also use OpenMP of course.)
- SPMD parallelization across SIMD lanes is *complementary* to TLP. Any thread can call an ISPC kernel independently.
- The *gang size* is small and is no more than twice the SIMD vector width on the target hardware.

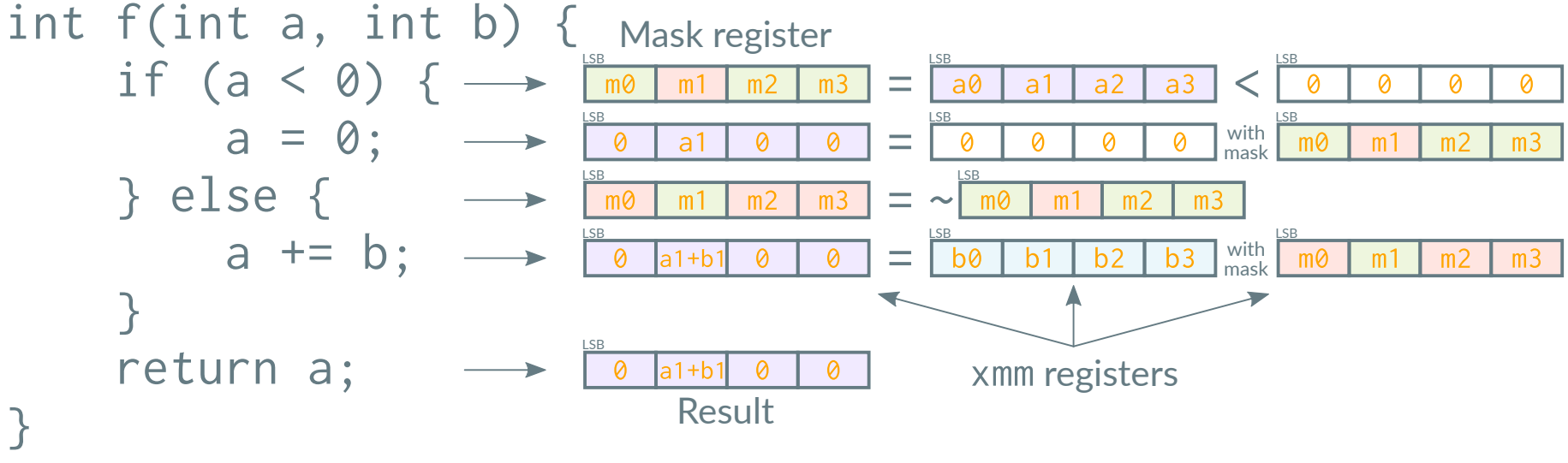
MAPPING "SPMD-ON-SIMD" TO HARDWARE

- One of the main challenges to exploit DLP with SIMD *efficiently* is to handle *divergent control flow* efficiently.
- We have not talked about this thoroughly when studying explicit vectorization with Intel intrinsics. It can be accomplished through *masking* of SIMD lanes and it further increases complexity when writing code.
- ***For example:*** what should happen at an `if`-statement when the condition *does not apply to all data elements* in the SIMD vector? Or at the end of a `for`-loop when the total iteration count does not divide evenly by the SIMD vector width?
- In ISPC's SPMD model you write vectorized code by putting yourself in the position of *one SIMD* lane. Thanks to this (SPMD) abstraction you can write vectorized code naturally and *productivity increases*. ISPC will take care of the (low-level) details.
- The burden of dealing with control flow divergence on the hardware level is shifted to the compiler by transforming *control flow* to *data flow* (see reading assignment).

MAPPING "SPMD–ON–SIMD" TO HARDWARE

Example control flow of a 4-way SIMD code in ISPC: (see reading assignment)

Gang size can be 4 or 8 for SSE (like in this example). If the gang size is 8, these operations are split into two SSE registers per variable.



- ISPC evaluates *both* execution paths of the `if`-branch.
- Control flow divergence is resolved with an *execution mask* on the hardware level.
- A team of SIMD lanes (program instances) is called a *gang*.
- When you write ISPC code you put yourself in the position of *one program instance* (any gang member).

SYNCHRONIZATION IN ISPC

- Because program instances can run freely in the SPMD model of ISPC, it must make some *synchronization* guarantees, since program instances map to SIMD vector registers → *"think parallel" applies here as well!*
- ISPC guarantees that the program counter and execution mask of program instances in gangs is ***maximally converged***. Maximal convergence is a synchronization guarantee which states that two program instances following *the same execution path* execute program statements *concurrently*. If their execution path *diverges*, it is guaranteed that they [program instances] re-converge at the *earliest point* where they could re-converge. *This guarantee stems from the notion of having **one** program counter per gang instead of one program counter per program instance.* See also https://ispc.github.io/ispc.html#gang-convergence-guarantees
- At the ISPC language level, *gangs execute in lockstep* and any side-effects (e.g. due to memory writes) are *visible to the gang* after the next sequence point in the program. *Examples of sequence points are* after every statement terminated with " ; ", before entering a function or when returning from a function.

SYNCHRONIZATION IN ISPC

- Consider the code from the previous slide:

```
1 int f(int a, int b)
2 {
3     if (a < 0) { // gang diverges (program instances take different execution paths)
4         a = 0;   // sequence point after ";"
5     } else {
6         a += b;  // gang is maximally converged after ";"
7     }
8     return a;
9 }
```

- Maximal convergence is not guaranteed for gangs across different threads. You must use *explicit* synchronization for this (e.g. `#pragma omp barrier` in OpenMP).
- This implicit synchronization in ISPC happens at sequence points.
- In CUDA, for example, synchronization is not as strict and must be enforced *by the programmer* with `__syncwarp()` across warps.
- After sequence points any memory writes are visible to the program instances in a gang.

ISPC LANGUAGE FEATURES

- ISPC syntax and basic functionality is based on C89 with some adoptions from C99 and C++ (examples include the ability to declare variables everywhere in a function, built-in `bool` type, references and function overloading).
- Arbitrary control flow is supported for ISPC program instances including: `if`, `switch`, `for`, `while`, do, `break`, `continue` and `return`.
- There are 4 important keywords in ISPC, two *type qualifiers* and two built-in variables to map gangs to data:

Type qualifiers:

`varying` (default, can be omitted)
One "private" data instance per program instance (SIMD-lane).

`uniform`
Scalar data instance, *shared* among program instances (SIMD-lanes).

Built-in variables:

`programIndex`
The index of the program instance (SIMD-lane).

`programCount`
The number of program instances in the gang.

BACK TO SYNCHRONIZATION IN ISPC: DEADLOCK?

- Consider the following ISPC code:

```
1 if (programIndex == 0) { // only SIMD-lane 0 executes this path
2     while (true) {}      // infinite loop
3 }
4 print("Hello, ISPC\n");  // print function from ISPC stdlib
```

- ISPC guarantees *maximal convergence* (think of a magnet that pulls program instances together) but gangs also execute in *lockstep*.
- ***What will the program instances with `programIndex` different than `0` do at line 2 in the code above?***
- ***Is the gang maximally converged after the `if`-block?***
- ***What happens in this case?***

REDUCTION EXAMPLE

Recall Problem 4 in HW4: reduction kernel with SIMD (SSE)

Explicit vectorization with intrinsics:

```
1  float sse_red_4(const float *ary, const size_t N)
2  {
3      const size_t simd_width = 16 / sizeof(float);
4      __m128 sum4 = _mm_setzero_ps();
5
6      // perform the reduction with correct stride:
7      for (size_t i = 0; i < N; i += simd_width) {
8          sum4 = _mm_add_ps(sum4, _mm_load_ps(ary + i));
9      }
10
11     alignas(16) float last[4];
12     _mm_store_ps(last, sum4);
13     return last[0] + last[1] + last[2] + last[3];
14 }
```

- `simd_width` plays the same role as `programCount`.
- `programIndex` does not show up in explicit vectorization since no SPMD model.
- Convenient `reduce_add` standard library function in ISPC (inter-lane communication).

ISPC equivalent:

```
1  export uniform float ispc_sse_red(const uniform float ary[],
2                                    const uniform size_t N)
3  {
4      varying float sum4 = 0.0; // could also omit `varying`
5      // assumes N evenly divides by programCount
6      for (uniform int i = 0; i < N; i += programCount) {
7          sum4 += ary[i + programIndex];
8      }
9      return reduce_add(sum4);
10 }
```

- The ISPC kernel must declare *scalar* variables as `uniform` (shared among program instances).
- The ISPC kernel requires the `export` keyword in order to be callable from C/C++ (C-linkage).

REDUCTION EXAMPLE

Recall Problem 4 in HW4: reduction kernel with SIMD (SSE)

ISPC equivalent:

```
1  export uniform float ispc_sse_red(const uniform float ary[],
2                                    const uniform size_t N)
3  {
4      float varying sum4 = 0.0; // could also omit `varying`
5      // assumes N evenly divides by programCount
6      for (uniform int i = 0; i < N; i += programCount) {
7          sum4 += ary[i + programIndex];
8      }
9      return reduce_add(sum4);
10 }
```

ISPC even better:

```
1  export uniform float ispc_sse_red(const uniform float ary[],
2                                    const uniform size_t N)
3  {
4      float varying sum4 = 0.0; // could also omit `varying`
5      // no assumptions on N!
6      foreach (i = 0 ... N) { // macro takes care of
7          sum4 += ary[i];     // remainder elements
8      }
9      return reduce_add(sum4);
10 }
```

- The `foreach` construct is used to specify parallel execution of program instances. It is similar in functionality to OpenMP's `omp for` construct.
- The `foreach` allows for iteration over multiple dimensions as well. *For example:*

```
1  foreach (ix = 0 ... Nx, iy = 0 ... Ny, iz = 0 ... Nz) // 3D parallel ISPC loop
```

- You *cannot* use a *break* or *return* statement in parallel loop constructs. The `continue` statement is allowed.
- Other forms of such loop constructs exist in ISPC as well → see https://ispc.github.io/ispc.html#iteration-statements

REDUCTION EXAMPLE: COMPILE ISPC CODE

Recall Problem 4 in HW4: reduction kernel with SIMD (SSE)

ISPC equivalent:

```
export uniform float ispc_sse_red(const uniform float ary[],
                                  const uniform size_t N)
{
    float varying sum4 = 0.0; // could also omit `varying`
    // assumes N evenly divides by programCount
    for (uniform int i = 0; i < N; i += programCount) {
        sum4 += ary[i + programIndex];
    }
    return reduce_add(sum4);
}
```

ISPC even better:

```
export uniform float ispc_sse_red(const uniform float ary[],
                                  const uniform size_t N)
{
    float varying sum4 = 0.0; // could also omit `varying`
    // no assumptions on N!
    foreach (i = 0 ... N) {
        sum4 += ary[i];
    }
    return reduce_add(sum4);
}
```

- ISPC code is great because you can easily generate machine code for many of the different Intel ISA extensions (e.g. SSE4, AVX or AVX2).
- To compile ISPC code, you must specify at least the following:
  1. The target architecture. *Examples* are `x86`, `x86-64` or `arm`
  2. The target ISA extension. *Here is where you determine the gang size. Example:* `sse4-i32x4` for SSE4 with a *mask size* of 32 and a *gang size* of 4. See the `--target` option in `ispc --help` for all the supported ISA targets.
- Compile the above ISPC code identical to the explicit 4-way SIMD code in HW4:

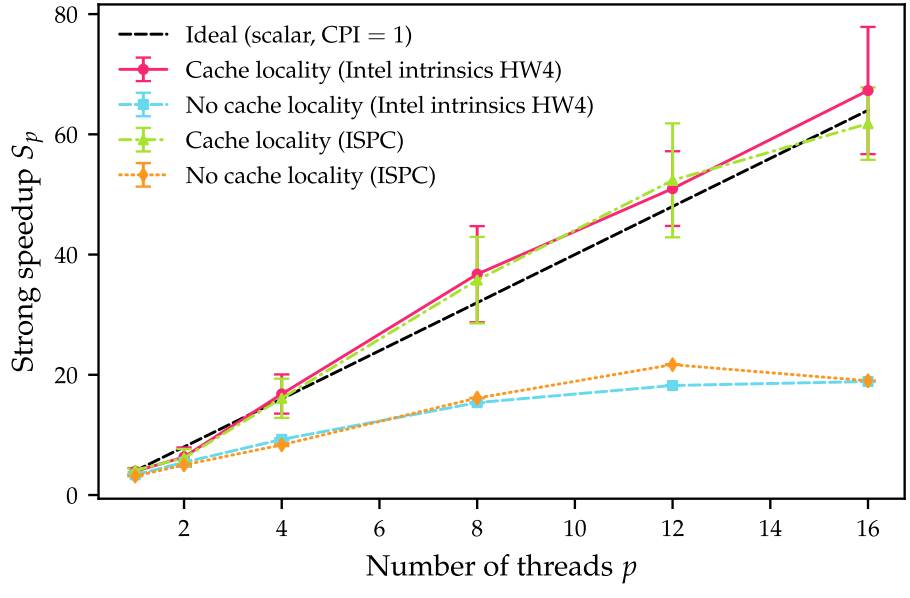
```
ispc -O3 --arch=x86-64 --target=sse4-i32x4 -o kernel.o kernel.ispc
```

REDUCTION EXAMPLE: RESULTS

Recall Problem 4 in HW4: Intel intrinsics (explicit) vs. ISPC (implicit)

- If the ISPC version of the reduction kernel of HW4 is compiled using the command on the previous slide the results of the figure on the right are obtained (single precision 4-way SIMD).
- In this case a *gang size of 4* will fit into one `xmm` register for single precision SSE.
- Results are the same as expected. ***Why is the speedup without cache locality less than ideal?***

Intel Xeon E5-2683v4 (Broadwell)



Reduction loop assembly (Intel intrinsics):

```
1  .L3:
2      addps   xmm1,XMMWORD PTR [rdi+rax*4] ; SSE SIMD add
3      add     rax,0x4 ; 64-bit integer arithmetic
4      cmp     rsi,rax ; check loop condition
5      ja      .L3
```

Reduction loop assembly (ISPC):

```
1  .L3:
2      movsxd rdx,edx   ; move 32-bit to 64-bit int (sign extend)
3      addps  xmm1,XMMWORD PTR [rdi+rdx*1] ; SSE SIMD add
4      add    eax,0x4   ; step 4 elements at a time
5      add    edx,0x10  ; move address by 16 byte
6      cmp    eax,ecx   ; check loop condition
7      jl     .L3
```

ISPC performs 32-bit addressing by default hence the sign-bit extension of `rdx` (`movsxd`).

REDUCTION EXAMPLE: RESULTS

Recall Problem 4 in HW4: Intel intrinsics (explicit) vs. ISPC (implicit)

What happens if we increase the ISPC gang size from 4 to 8 (`--target=sse4-i32x8`)?

Reduction loop assembly (Intel intrinsics):

```
.L3:
    addps   xmm1,XMMWORD PTR [rdi+rax*4] ; SSE SIMD add
    add     rax,0x4 ; 64-bit integer arithmetic
    cmp     rsi,rax ; check loop condition
    ja      .L3
```

(same as on previous slide)

Reduction loop assembly (ISPC):

```
.L3:
    movsxd rdx,edx
    addps   xmm3,XMMWORD PTR [rdi+rdx*1+0x10]
    addps   xmm2,XMMWORD PTR [rdi+rdx*1]
    add     eax,0x8  ; step 8 elements at a time!
    add     edx,0x20 ; move address by 32 byte!
    cmp     eax,ecx
    jl      .L3
```

If we manually unroll the reduction loop in the ISPC kernel we get even more ILP:

```
export uniform float ispc_sse_red(const uniform float ary[],
                                  const uniform size_t N)
{
    float varying sum0 = 0.0; float varying sum1 = 0.0;
    float varying sum2 = 0.0; float varying sum3 = 0.0;
    // assumes N evenly divides by 4 * programCount
    for (uniform int i = 0; i < N; i += 4 * programCount) {
        sum0 += ary[i + 0 * programCount + programIndex];
        sum1 += ary[i + 1 * programCount + programIndex];
        sum2 += ary[i + 2 * programCount + programIndex];
        sum3 += ary[i + 3 * programCount + programIndex];
    }
    return reduce_add(sum0) + reduce_add(sum1)
         + reduce_add(sum2) + reduce_add(sum3);
```

We have just doubled ILP! (and increased register pressure, but OK...) → *this is **very** nice!*

```
.L3: ; this is impressive code!
    lea     edx,[rcx-0x60]
    movsxd rdx,edx
    addps   xmm4,XMMWORD PTR [rdi+rdx*1+0x10]
    addps   xmm3,XMMWORD PTR [rdi+rdx*1]
    lea     edx,[rcx-0x40]
    movsxd rdx,edx
    addps   xmm5,XMMWORD PTR [rdi+rdx*1+0x10]
    addps   xmm2,XMMWORD PTR [rdi+rdx*1]
    lea     edx,[rcx-0x20]
    movsxd rdx,edx
    addps   xmm6,XMMWORD PTR [rdi+rdx*1+0x10]
    addps   xmm0,XMMWORD PTR [rdi+rdx*1]
    movsxd rcx,ecx
    addps   xmm7,XMMWORD PTR [rdi+rcx*1+0x10]
    addps   xmm1,XMMWORD PTR [rdi+rcx*1]
    add     eax,0x20
    sub     ecx,0xffffff80
    cmp     eax,esi
    jb      .L3
```

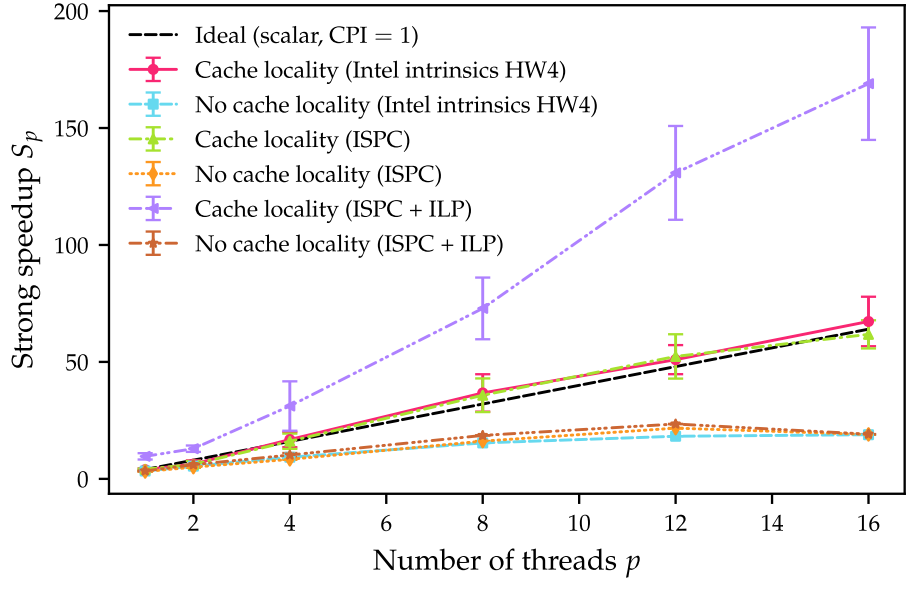
REDUCTION EXAMPLE: RESULTS

Recall Problem 4 in HW4: Intel intrinsics (explicit) vs. ISPC (implicit)

DLP with ISPC + ILP + temporal locality: 156x speedup → serious CPU cooling required!

Comparison between 1 and 16 threads on one Intel Xeon E5-2683v4 CPU:

```
1  TESTING SIZE N = 32768 -- fits into L1 cache
2  4-way SSE       : 1 thread single precision; -- speedup: 4.06
3  ISPC gang size 8: 1 thread single precision; -- speedup: 10.76
4  4-way SSE       : 16 threads single precision; -- speedup: 61.82
5  ISPC gang size 8: 16 threads single precision; -- speedup: 155.84
6  TESTING SIZE N = 16777216 -- too large for L1 cache
7  4-way SSE       : 1 thread single precision; -- speedup: 3.34
8  ISPC gang size 8: 1 thread single precision; -- speedup: 3.52
9  4-way SSE       : 16 threads single precision; -- speedup: 19.25
10 ISPC gang size 8: 16 threads single precision; -- speedup: 24.13
```



Superscalar execution because of high ILP, out-of-order execution, multiple execution units, data prefetching and a *full* pipeline.

Efficient use of the cache is of *utmost* importance! Without temporal locality (e.g. reduction algorithm in real life) all of the ILP beauty generated by ISPC does not bring you anything! See the last legend entry → ***bandwidth is your true enemy***.

ISPC WORKFLOW

ISPC is used to write optimized SIMD code for compute kernels. Such kernels are usually written in a source file (compilation unit) which is then *linked* to the application code.

1. You begin with a sequential baseline code that should be optimized for cache locality.
2. Identify the performance critical function(s) in your code using profiling techniques. For example, you can use GNU profiler `gprof` (https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html), the Linux native profiling utilities (`perf`, `strace`) or you can manually add timer calls in your code if necessary (you should prefer a profiler).
3. Rewrite the performance critical code using ISPC.
4. Compile the ISPC kernel(s) to object code and link the code to your application (2 step process). *The application code may use other parallel programming models like OpenMP and/or MPI.*

ISPC ADVANTAGES

- Simple and small C/C++ language extension. No API needed like in the case of Intel intrinsics or MPI. You call ISPC kernels in C++ through the `extern "C"` export, but you can also call C++ functions inside ISPC kernels (it goes both ways).
- Trivial to increase ILP with ISPC by increasing the gang size and additional unrolling of code.
- The "SPMD-to-SIMD" programming model provides a serious productivity boost. ISPC code is portable to different architectures and ISA extensions. *Similar model is exploited in Nvidia CUDA*.
- ISPC provides convenience functions to perform common inter-lane SIMD operations like `reduce_add`, `reduce_max`, `shuffle` math functions and others. See https://ispc.github.io/ispc.html#the-ispc-standard-library.
- ISPC focus is on DLP → ISPC kernels can easily be combined with threading models like OpenMP to exploit TLP.

POINTERS AND ARRAYS IN ISPC

The `uniform` and `varying` type qualifiers in ISPC can cause confusion:

- `uniform`: scalar storage that is *shared* among all the program instances.
- `varying`: separate (*private*) storage per program instance.

- Variables are `varying` in ISPC by default.
- The `uniform` qualifier exists to help the compiler generate better code which also utilizes the scalar execution units (e.g. constants, loop counters or memory addressing).
- These two type qualifiers apply to pointers (and arrays) for *both* the pointer itself and the data the pointer points to.

There are 4 kinds of pointers in ISPC:

- `uniform float * uniform ptr;`
- `varying float * uniform ptr;`
- `uniform float * varying ptr;`
- `varying float * varying ptr;`

Arrays:

- `float a[10];` is the same as `varying float a[10];` (one array per program instance).
- Use `uniform float a[]` for function arguments (one array shared among program instances → DLP).

ISPC STANDARD LIBRARY

- The ISPC compiler comes with a standard library that provides many useful functions for math, reductions, logical operations, assertions, random numbers or cross-lane (cross-program instance) operations (shuffles) among others.
- These functions help with vectorizing irregular parts of your code (these are the tedious parts when vectorizing explicitly!).
- A few examples: (https://ispc.github.io/ispc.html#the-ispc-standard-library)

Math:

- `float abs(float a)`
- `uniform float abs(uniform float a)`
- `float min(float a, float b)`
- `float sqrt(float v)`
- `float rsqrt_fast(float v)`
- `float cos(float x)`
- `float log(float x)`

Cross-lane:

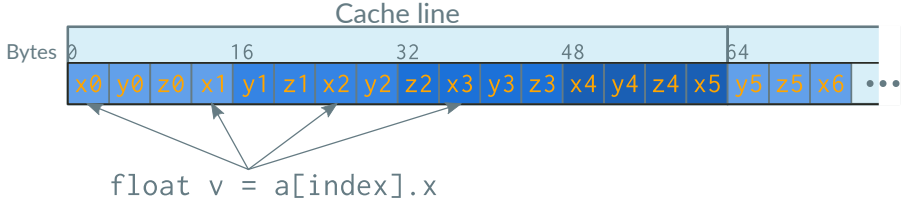
- `float broadcast(float value, uniform int index)`
- `float rotate(float value, uniform int offset)`
- `int32 shuffle(int32 value, int permutation)`
- `uniform float extract(float x, uniform int i)`
- `uniform int32 reduce_add(int16 x)`

ISPC SUPPORT FOR DATA LAYOUTS

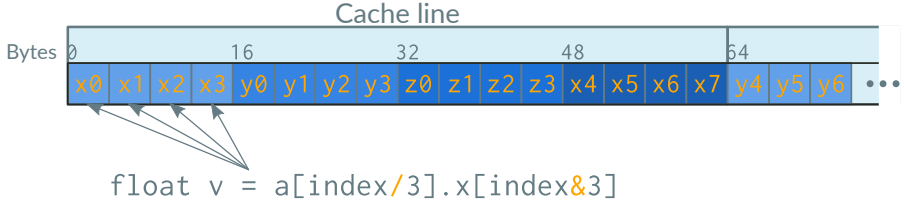
First sentence in Section 5.2 of reading assignment:

> *It is well known that the standard C/C++ layout in memory for an "array of structures" (AoS) leads to sub-optimal performance for SIMD code.*
>
> *M. Pharr and W. R. Mark, 2012*

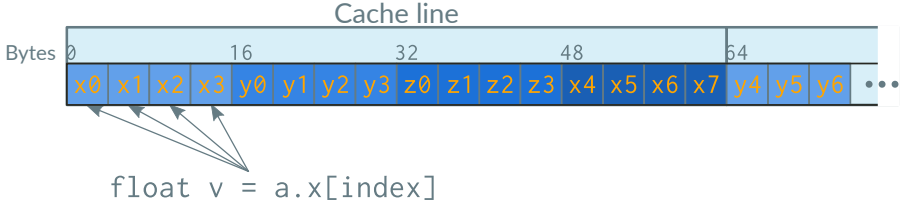
Recall AoS layout from last lecture:



Hybrid SoA:



Short SoA



Example ISPC code: bad spatial locality

```
1 struct Foo { float x, y, z; };
2 uniform Foo a[...] = {/* initialize */};
3 int index = ...; // varying int
4 float v = a[index].x;
```

Example ISPC code: tedious index computation

```
1 struct Foo4 { float x[4], y[4], z[4]; };
2 uniform Foo4 a[...] = {/* initialize */};
3 int index = ...; // varying int
4 float v = a[index / 4].x[index & 3];
```

Example ISPC code: using soa<4> keyword

```
1 struct Foo { float x, y, z; };
2 soa<4> struct Foo a[...] = {/* initialize */};
3 int index = ...; // varying int
4 float v = a[index].x; // treat as if AoS
```

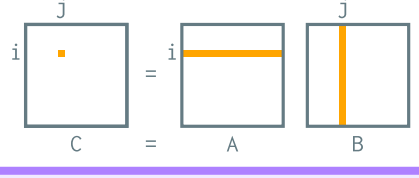
EXAMPLE: GEMM WITH ISPC AND EIGEN

Consider square matrices of size 1024 times 1024:

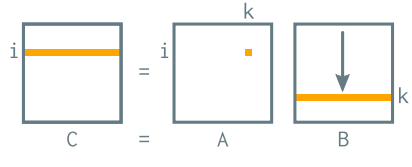
- We are interested in comparing the the performance of a general matrix-matrix multiplication:

```
1  void gemm(const Real *A, const Real *B,
2            Real *__restrict__ C,
3            const int p, const int r, const int q)
4  {
5      for (int i = 0; i < p; ++i) {
6          for (int j = 0; j < q; ++j) {
7              for (int k = 0; k < r; ++k) {
8                  C[i*q + j] += A[i*r + k] * B[k*q + j];
9              }
10         }
11     }
12 }
```

No spatial locality:



With spatial locality:



- A version optimized for cache could either *swap the innermost loop with the middle loop* or use a *tiled* iteration:

```
1  void gemm_tile(const Real *A, const Real *B,
2                 Real *__restrict__ C,
3                 const int p, const int r, const int q)
4  {
5      for (int I = 0; I < p; I += _TILE_) {
6      for (int J = 0; J < q; J += _TILE_) {
7      for (int K = 0; K < r; K += _TILE_) {
8          for (int i = I; i < I + _TILE_; ++i) {
9          for (int k = K; k < K + _TILE_; ++k) {
10         for (int j = J; j < J + _TILE_; ++j) {
11             C[i*q + j] += A[i*r + k] * B[k*q + j];
12         }
13         }
14         }
15     }
16     }
17     }
18 }
```

CPU	Naive	Two inner loops swapped	Tiled
AMD Opteron 6376 (2012, cluster)	63622.9 ms	2787.1 ms (22.8x faster)	3761.16 ms (16.9x faster)
Intel Xeon E5-2683v4 (2016, cluster)	2572.8 ms	668.6 ms (3.8x faster)	840.3 ms (3.1x faster)
AMD Ryzen 7 PRO 4750U (2020, laptop)	6714.0 ms	553.1 ms (12.1x faster)	612.2 ms (11.0x faster)

EXAMPLE: GEMM WITH ISPC AND EIGEN

ISPC implementation with tiling and manual loop unrolling:

```
export void gemm_ispc(const uniform Real A[], const uniform Real B[],
    uniform Real C[], uniform int p, uniform int r, uniform int q)
{
    uniform Real tile[16][_TILE_]; // scratch storage
    for (uniform int i = 0; i < p; i += 16) {
        for (uniform int j = 0; j < q; j += _TILE_) {
            foreach (th = 0 ... _TILE_) { // initialize scratch to zero
                tile[0][th] = (Real)0.0;
                // ...repeated
                tile[15][th] = (Real)0.0;
            }

            // start of inner product
            for (uniform int k = 0; k < r; k += programCount) {
                const Real Aik_v0 = A[(i + 0) * r + k + programIndex];
                // ...repeated
                const Real Aik_v15 = A[(i + 15) * r + k + programIndex];
                for (uniform int tv = 0; tv < programCount; ++tv) {
                    const Real Aik_u0 = shuffle(Aik_v0, tv);
                    // ...repeated
                    const Real Aik_u15 = shuffle(Aik_v15, tv);
                    foreach (th = 0 ... _TILE_) { // SPMD dot-products
                        const float Bik = B[(k + tv) * q + j + th];
                        tile[0][th] += Aik_u0 * Bik;
                        // ...repeated
                        tile[15][th] += Aik_u15 * Bik;
                    }
                }
            }

            foreach (th = 0 ... _TILE_) { // SPMD store
                C[(i + 0) * q + j + th] = tile[0][th];
                // ...repeated
                C[(i + 15) * q + j + th] = tile[15][th];
            }
        }
    }
}
```

Compilation:

1. Compile the kernel to AVX2 machine code

```
ispc -O3 --arch=x86-64 --target=axv2-i32x8 \
    -o gemm_avx2.o -h gemm_avx2.h gemm.ispc
```

   - The `-o` option is the same as for GCC.
   - The target ISA in this example is AVX2 with a gang size of 8.
   - The `-h` option generates a *header file* that can be used to include in your driver code.
   - The last input is the ISPC source code.

2. In the second step you compile your application code (which may include the header generated with the `-h` flag above) and link to the ISPC object code:

```
g++ -O2 -o main main.cpp gemm_avx2.o
```

   The application code will then call the AVX2 ISPC version of the GEMM kernel.

EXAMPLE: GEMM WITH ISPC AND EIGEN

Results without SIMD:

CPU	Naive	Two inner loops swapped	Tiled
AMD Opteron 6376 (2012, cluster)	63622.9 ms	2787.1 ms (22.8x faster)	3761.16 ms (16.9x faster)
Intel Xeon E5-2683v4 (2016, cluster)	2572.8 ms	668.6 ms (3.8x faster)	840.3 ms (3.1x faster)
AMD Ryzen 7 PRO 4750U (2020, laptop)	6714.0 ms	553.1 ms (12.1x faster)	612.2 ms (11.0x faster)

Results with AVX2 ISPC code relative to the Tiled version without SIMD:

CPU	Tiled	ISPC	Eigen	Eigen (optimized)
Intel Xeon E5-2683v4 (2016, cluster)	835.2 ms	*79.5* ms (10.5x faster)	94.2 ms (8.9x faster)	26.9 ms (31.0x faster)
AMD Ryzen 7 PRO 4750U (2020, laptop)	594.0 ms	*45.7* ms (13.0x faster)	60.8 ms (9.8x faster)	25.7 ms (23.1x faster)

Another reference kernel using the Eigen library is shown in the table above. This kernel simply computes:

```
1  void gemm_eigen(const Real *A, const Real *B,
2      Real *__restrict__ C,
3      const int p, const int r, const int q)
4  {
5      // Matrix wrappers (similar to lab 5)
6      const EMat<Real> Ap(const_cast<Real *>(A), p, r);
7      const EMat<Real> Bp(const_cast<Real *>(B), r, q);
8      EMat<Real> Cp(C, p, q);
9      Cp.noalias() += Ap * Bp; // GEMM
10 }
```

Eigen ships with various kernel implementations for its internal operations. Highly optimized versions with *blocking for all cache levels* can be compiled depending on the architecture you target. With GCC you can pass the `-march=broadwell` flag when targeting Broadwell architectures. For AMD Zen version 2 it would be `-march=znver2`. See `man gcc` for all supported architectures.

RECAP

- Explicit vectorization with Intel intrinsics (for example) is tedious and laborious.
- ISPC provides a SPMD programming model that helps with writing SIMD aware code (exploit DLP). *Nvidia CUDA is very similar to this programming model*.
- ISPC is based on the LLVM framework and can generate impressive results that easily outperform the compilers auto-vectorization capabilities.
- To start writing code in ISPC only four new keywords are necessary. It is important that they are well understood (especially the difference between the type qualifiers `uniform` and `varying`).
- ISPC can target many different architectures by writing only one code, i.e., it is a domain specific language (DSL) for domain specific architectures (DSA).

Further reading:

- ISPC documentation: https://ispc.github.io/ispc.html
- Intel Intrinsics Guide: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- *"Intel© 64 and IA-32 Architectures Software Developer's Manual"*, Volume 2, Instruction Set Reference, pdf
- Chapter 4 in *"Computer Architecture"*, J. Hennessy and D. Patterson, Morgan Kaufmann 2019