# SPMD - Intel SIMD programming with ISPC

Joao Barbosa

Week 3, March 2021

# What is ISPC

- **I**ntel **S**PMD **P**rogram **C**ompiler
  - Single Program Multiple Data
- Compiler and Language to write vector code
- C Based language
- Simple to use and integrate into existing code
- **It is not** a **Auto-Vectorizing** compiler
  - **Vectors are explicitly declared in the type system**
  - **Developers explicitly declare what is scalar and vector**

# Why ISPC?

- Exploiting parallelism is critical for performance
  - Task parallelism
  - SIMD parallelism
- Can be done with **Intrisics**
  - **Intrinsics** are platform specific
  - Hard to understand and get right
- Intel ISPC
  - Easier to get flops without beging a “ninja” programmer
  - High-level programming (easier to develop and maintain)

# ISPC Language

- C Based
- Code looks sequential but is parallel
- Two new keywords:
  - uniform : scalar value
  - varying : vector value
- *foreach* vector iteration

```
export void rgb2grey(uniform int N,
                     uniform float R[],
                     uniform float G[],
                     uniform float B[],
                     uniform float grey[]) {
    foreach( i = 0 … N) {
        varying float vGray = 0.3f * R[i] + 0.59f * G[i]
                                         + 0.11f * B[i];
        grey[i] = vGrey;
    }
}
```

# ISPC Integration

- ISPC compiler produces everything required for integration with C/C++ code
- C/C++ header file
  - Contains the API/function call for each *kernel* you have written
  - Contains any datastructure defined in your ISPC kernel
- Object files to link against
- No runtime or verbose API
  - Typical of OpenCL for instance

# Good because …

- Programmers no longer need to know the underlaying ISA
- Reduced development and maintenance cost
- Increased optimization reach
- Increased performance
  - SSE - ~4x
  - AVX2 - ~6x

# Vector Loops

***foreach***

```
export uniform int sum (uniform int val[], uniform int N) {
    varying int sum = 0;
    foreach( i = 0 ... N) {
        sum += val[i];
    }
    return reduce_add(sum);
}
```

***for loop***

```
export uniform int sum(uniform int val[],uniform int N) {
    varying int sum = 0;
    for(varying int i = programIndex; i < N;
                i+= programCount) {
        sum += val[i];
    }
    return reduce_add(sum);
}
```

# Vector Loop

***foreach***

- SIMD for loop
- Iterates over chunks of SIMD width
- UNMASKED main body for all SIMD lanes
- Masked tail body for when some SIMD lanes are disabled
- *foreach* can be N dimensional, with each dimension being a varying

***for loop***

- A *for* loop with a varying index will use masking in the loop body
- A *for* loop with a uniform index will have no masking

```
struct vec4 {
    float v[4];
};
void vec4MultipleAdd(varying vec4& result,
                     const varying vec4& A,
                     const varying vec4& B,
                     const varying vec4& C) {

for(uniform int i=0; i < 4; i++) {
    result.v[i] = A.v[i] * B.v[i] + C.v[i];
}
}
```

```
void vec4MultipleAdd(uniform vec4& result,
                     const uniform vec4& A,
                     const uniform vec4& B,
                     const uniform vec4& C) {

for(uniform int i=0; i < 4; i++) {
    result.v[i] = A.v[i] * B.v[i] + C.v[i];
}
}
```

```
void vec4MultipleAdd(uniform vec4& result,
                     const uniform vec4& A,
                     const uniform vec4& B,
                     const uniform vec4& C) {

foreach(i = 0 ... 4) {
    result.v[i] = A.v[i] * B.v[i] + C.v[i];
}

}
```

***foreach_active***

- Serializes over each active SIMD lane

- Many uses:
  - Atomic operations
  - Custom reductions
  - Calls to uniform functions

```
export uniform int sum(uniform int val[], uniform int N) {
    uniform int sum[progamCount];
    sum[programIndex] = 0;
    for(varying int i = programIndex; i < N; i+= programCount)
{
        sum[programIndex] += val[i];
    }
    uniform int ret = 0;
    foreach_active(j) {
        ret += sum[j];
    }
}
```

# Axis of parallelism

```
struct vec4 {
    float v[4];
};

// Varying - 4 vector FMAs
void vec4multadd0(varying vec4& result, const varying vec4& A, const varying vec4& B,
 const varying vec4& C) {
        for(uniform int i=0; i < 4; i++) {
            result.v[i] = A.v[i] * B.v[i] + C.v[i];
        }
}

// Uniform - 4 scalar FMAs
void vec4multadd1(uniform vec4& result, const uniform vec4& A, const uniform vec4& B,
 const uniform vec4& C) {
        for(uniform int i=0; i < 4; i++) {
            result.v[i] = A.v[i] * B.v[i] + C.v[i];
        }
}

// Uniform - 1 vector FMAs
void vec4multadd2(uniform vec4& result, const uniform vec4& A, const uniform vec4& B,
 const uniform vec4& C) {
        foreach( i = 0 ... 4) {
            result.v[i] = A.v[i] * B.v[i] + C.v[i];
        }
}
```

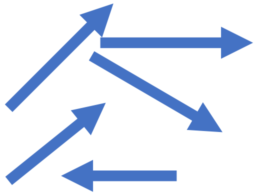
# Uniform vs Varying structures

```
struct vec3f {
    float x,y,z;
};

struct ray {
    vec3f ori;
    vec3f dir;
    float tnear;
    float tfar;
};
```

## Uniform

## Varying



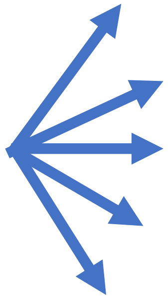
# Uniform vs Varying structures

```
struct vec3f {
    float x,y,z;
};

struct ray {
    uniform vec3f ori;
    vec3f dir;
    float tnear;
    float tfar;
};
```

**Uniform**

**Varying**



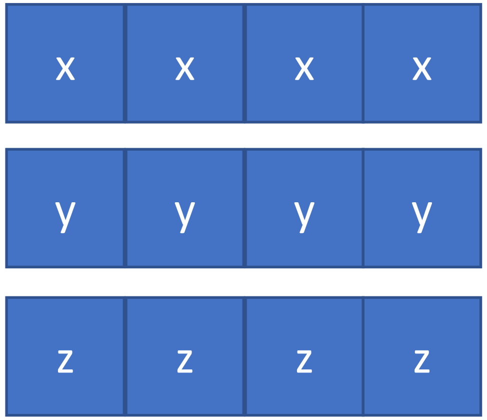
# Uniform vs Varying data layout

```
struct vec3f {
    float x,y,z;
};

struct ray {
    vec3f ori;
    vec3f dir;
    float tnear;
    float tfar;
};
```

**Uniform**


**Varying**

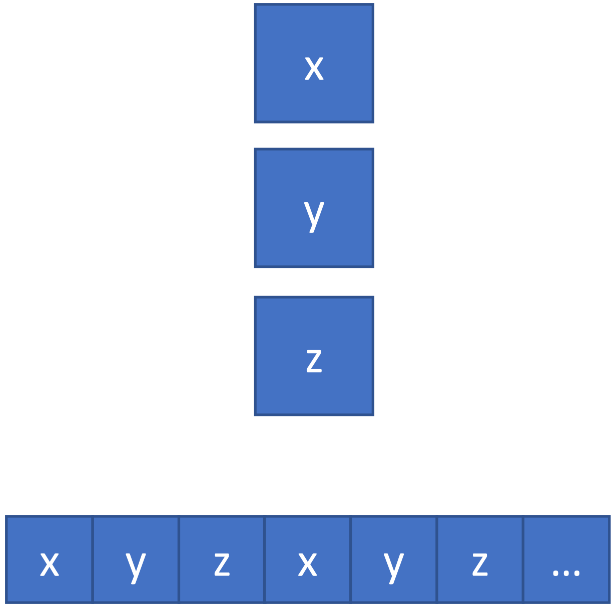


# Uniform vs Varying data layout

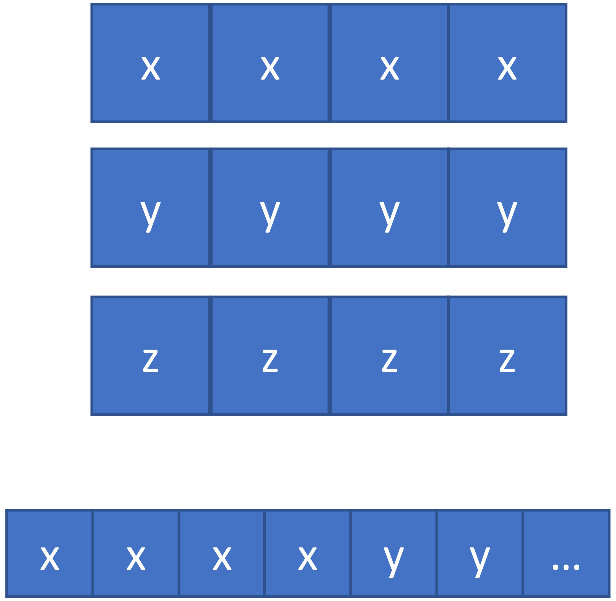
```
struct vec3f {
    float x,y,z;
};

struct ray {
    vec3f ori;
    vec3f dir;
    float tnear;
    float tfar;
};
```

## Uniform



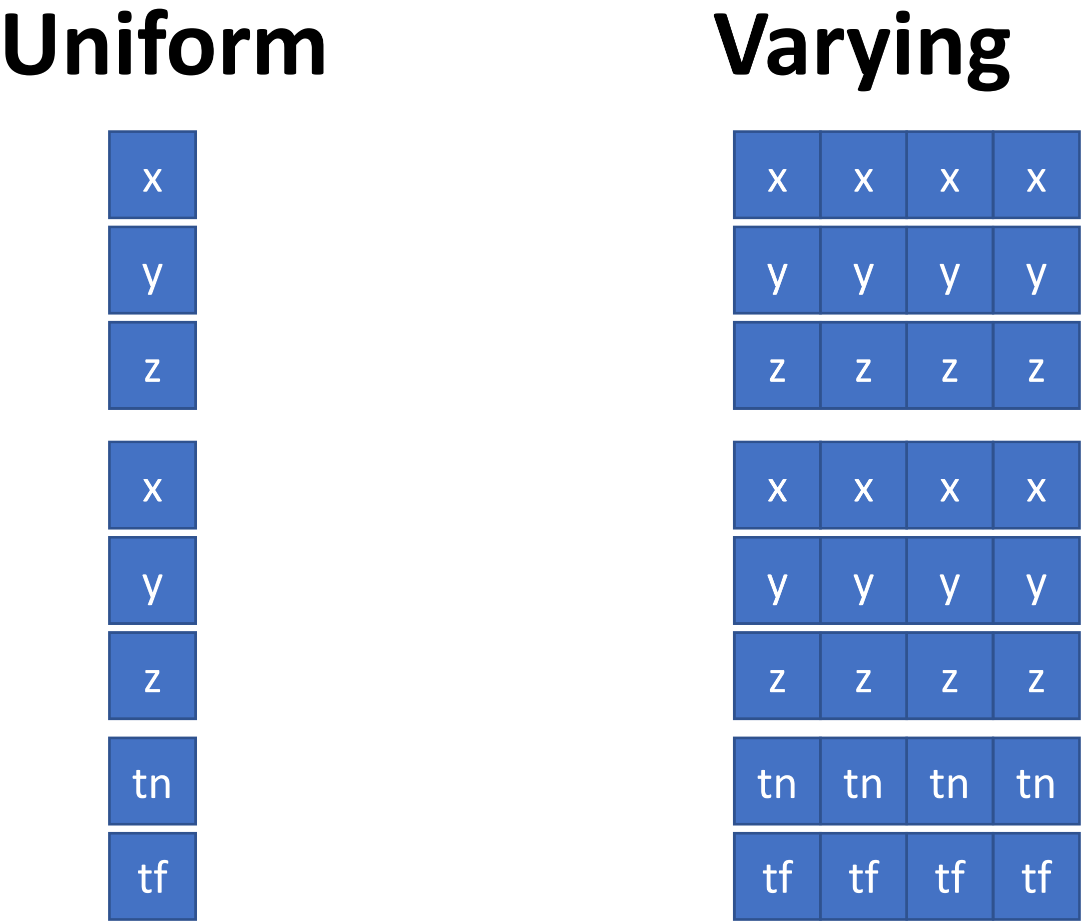

## Varying



# Uniform vs Varying data layout

```
struct vec3f {
    float x,y,z;
};

struct ray {
    uniform vec3f ori;
    vec3f dir;
    float tnear;
    float tfar;
};
```



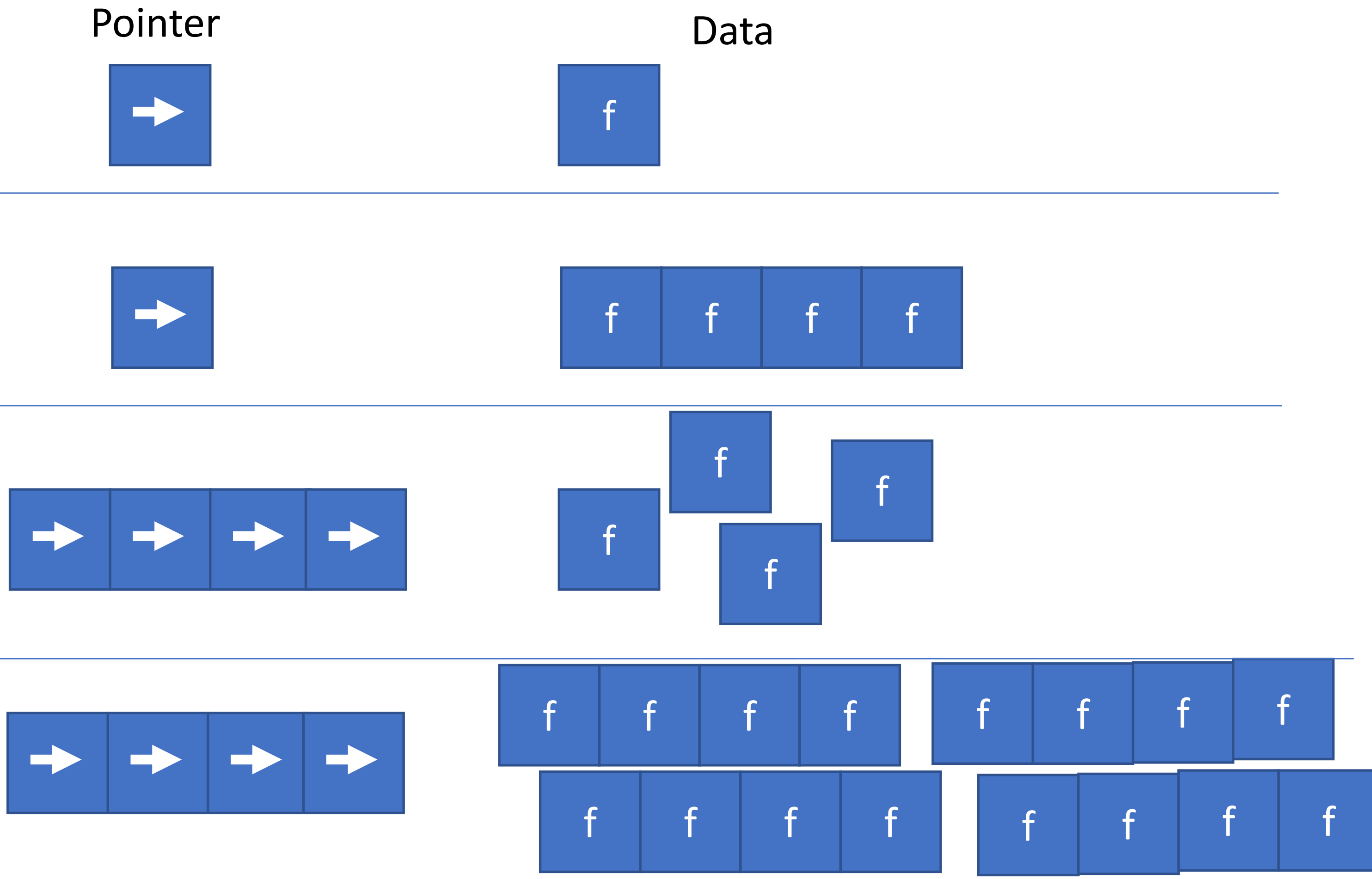
# Uniform vs Varying Pointers

uniform float* uniform vPtr

uniform float* varying vPtr

varying float* uniform vPtr

varying float* varying vPtr



# Data transposition

```
varying gatherrays(uniform * ray uniform rays, varying int index) {
    varying ray rrays = rays[index];
    return rrays;
}
```

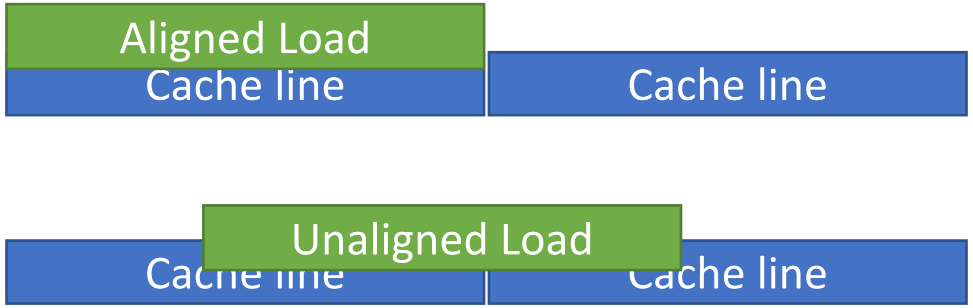
- ISPC will automatically transpose your array of structures to a structure of arrays and back
  - Useful to block copying uniform structures into varying
  - It will just works
  - But there may be faster alternatives…

# Gather / Scatter

- Vector reads/writes to non-contiguous memory
  - AVX2 onwards supports an optimized **gather** operation
  - AVX512 supports an optimized **scatter** operation
  - ISPC will use them if available
- ISPC emmits warnings when it finds scatter/gathers
  - #pragma ignore warning(perf)
- Gather performance has improved over successive generations
  - But there can be faster alternatives especially if there is cache line locality
- aos_to_soa helpers
  - For float3/float4

# Aligned Memory

- Load/Stores can be aligned or unaligned
  - There are specific instructions for each type
- Historically this had a performance impact
  - Unaligned load/stores may straddle cacheline
  - Newer intel hardware has remove/minimized this impact
- Aligment needs to be register vector width
  - SSE – 16 bytes
  - AVX2 – 32 bytes
  - AVX512 – 64 bytes
- Simple to enable in ISPC
  - --opt=force-aligned-memory

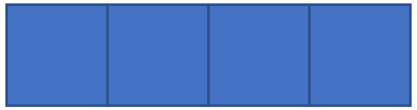


# Control divergence

```
void test(varying bool performOP1) {

    if(performOP1)
        doExpensiveOp1()
    else
        doExpensiveOp2()

}
```



# Control divergence

```
void test(varying bool performOP1) {

    if(performOP1)
        doExpensiveOp1()
    else
        doExpensiveOp2()

}
```

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |

# Control divergence

```
void test(varying bool performOP1) {

    if(performOP1)
        doExpensiveOp1()
    else
        doExpensiveOp2()

}
```

# Control divergence

```
void test(varying bool performOP1) {

    if(performOP1)
        doExpensiveOp1()
    else
        doExpensiveOp2()

}
```

# Control divergence

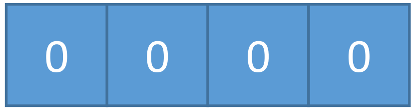
```
void test(uniform bool performOP1) {

    if(performOP1)
        doExpensiveOp1()
    else
        doExpensiveOp2()

}
```


doExpensiveOp1()

or




doExpensiveOp2()

# C/C++ Interfacing

- Mapping data to varyings
  - Input data is an array of uniforms
  - These can be copied directly to varyings by using a varying index
    - Such as programIndex
  - They can be cast to a varying pointer and dereferenced
  - Application can pass a 'fake' varyings which still generate SIMD code

# Calling back to C

- Just like in C/C++ there are times when you need to call external code
- ISPC supports it as long as it is a 'C' function

```
extern "C" void myUniformDebugFunc(uniform int active_simd_lane);

void func1(varying bool flag) {
    if(flag) {
        foreach_active(i) {
            myUniformDebugFunc(i);
        }
    }
}

extern "C" void myVaryingDebugFunc(uniform float* uniform data, uniform int count);

void func2(varying float data) {
    uniform float** uniform pdata = (uniform float** uniform)&data;
    myVaryingDebugFunc(*pdata,programCount);
}
```

# Autodispatch/Choosing the right target

- ISPC supports compiling to multiple targets at once
  - Only 1 target per ISA
  - The auto dispatch will choose the highest supported compiled target that a platform supports
  - Manual dispatching in the future

- Compile for all the main targets
  - SSE4, AVX2,AVX512
  - This will allow the best performing ISA to run on your system

--target=sse4-i32x4,avx2-i32x8,avx512skx-i32x16 (,...)

# ISPC has a STD library

- ISPC provides a reach STD library
  - Logical operators
  - Bit operations
  - Math
  - Clamping and Staturation arithematic
  - Transcendental operations
  - RNG (Not the fastest)
  - Mask/Cross lane operations
  - Reductions
  - (...)