ISPC IN UNREAL ENGINE 4: A PEEK BEHIND THE CURTAIN

Jeff Rous, Senior Developer Relations Engineer, Intel

Michael Lentine, Lead Physics Programmer, Epic Games

Martin Wilson, Senior Animation Programmer, Epic Games

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Your costs and results may vary.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Agenda

Why Are We Here?

What Is ISPC?

ISPC Programming

Unreal and ISPC

Chaos and ISPC

Animation and ISPC

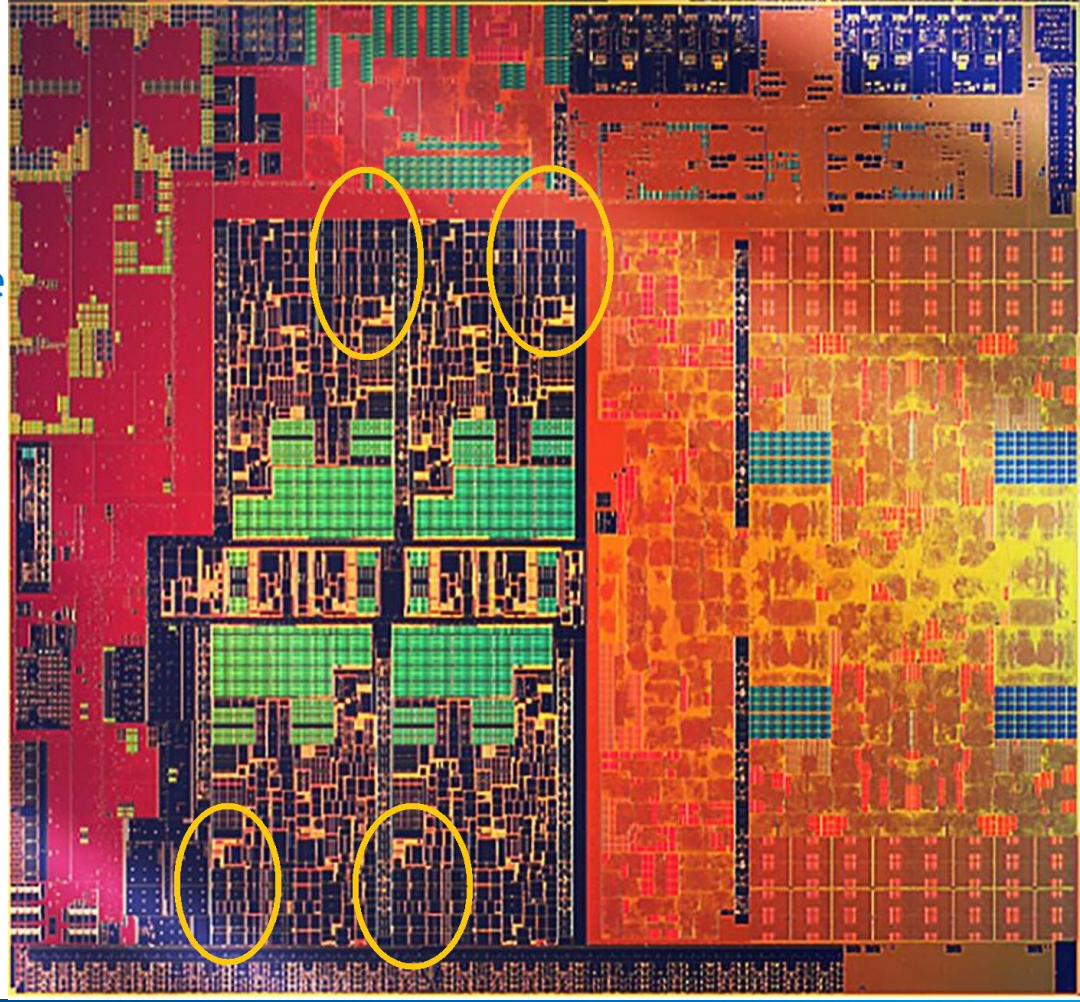
Wrap up

Why are we here?

- Exploiting Parallelism was essential for obtaining peak performance in Chaos and Animation, even on a modern high end system
  - Task Parallelism : Multithreading, Multi-core
  - **SIMD Parallelism : SIMD vector instructions**
- C++ doesn't cut it for high performance code. Auto vectorization is difficult to control. There's no time to create anything if everyone's learning how to write vector intrinsics.
  - Make it easier to get all the FLOPs without being a ninja programmer
  - Answer? **Intel® SPMD Program Compiler (ISPC)!**

Where are we?

- Ice Lake U (10th Gen) quad core CPU
- Yellow circles represent CPU execution units where the magic happens!



WHAT IS ISPC?

What is ISPC?

- The Intel SPMD Program Compiler
  - SPMD == Single Program, Multiple Data programming model
- It's a compiler and a language for writing vector (SIMD) code.
  - Open-source, LLVM-based language and compiler for many SIMD architectures.
  - Generates high performance vector code for many vector ISAs.
    - SSE/AVX/AVX2/AVX-512/NEON… (experimental)
- The language looks very much like C
  - Simple to use and easy to integrate with existing codebase. C function calls

Why ISPC?

Easy to speed up existing C++ code using SIMD quickly

Intrinsics are hard and instruction set specific. Adding AVX would mean another permutation. Unreal uses SSE2 intrinsics in a platform abstraction

Integration into engine benefits all games built on it

Works everywhere Unreal does (Win, Mac, Linux, PS4, Xbox, ARM)

Proven technology via other engine integrations

- Embree (Lightmass static lighting)
- ISPC Texture Compressor (BC6H / BC7 / ASTC)

ISPC PROGRAMMING

ISPC Programming Model

ISPC is ***not*** an “autovectorizing” compiler!

- It does not generate vector code by analyzing and transforming scalar loops.
  - ICC, Clang/LLVM, GCC, MSVC
- ISPC is more of a WYSIWYG vectorizing compiler
  - The programmer tells ISPC what’s vector and what’s scalar
  - Vector types are ***explicit***, not discovered
- Clean separation of SIMD vector and task programming models.
- Works on existing C/C++ memory allocations, data buffers.

What does the language look like?

- It looks very much like C, so it's easy to read and understand
- Code looks sequential, but executes in parallel
  - Easily mixes scalar and vector computation
- Explicit vectorization using two new ISPC language keywords, **uniform** and **varying**
  - Again, ISPC is ***not*** an auto-vectorizing compiler.

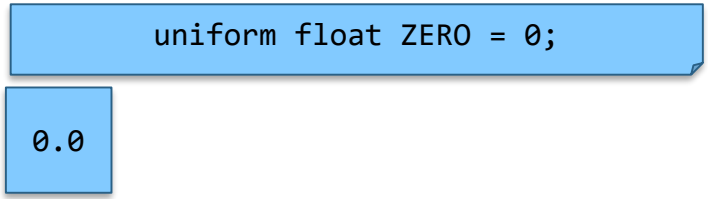
```
export void rgb2grey(uniform int N,
                     uniform float R[],
                     uniform float G[],
                     uniform float B[],
                     uniform float grey[])
{
  foreach (i=0 ... N)
  {
     grey[i] = 0.3f*R[i] + 0.59f*G[i] + 0.11f*B[i];
  }
}
```

- *It's basically shader programming for the CPU!*

The key concept -- uniform vs. varying

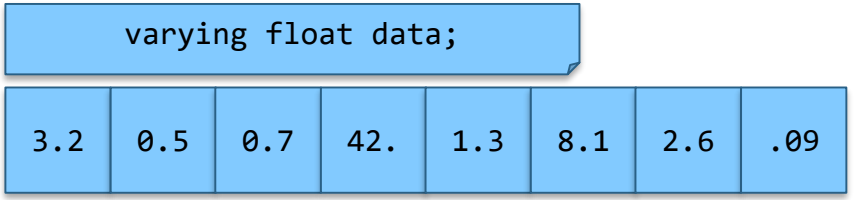
Uniform

- Scalar data
  - Results in a scalar register (eax, ebx, ecx, etc...)
  - All SIMD lanes share the same value.



Varying

- Vector data
  - Results in a SIMD vector register (XMM, YMM, ZMM, etc.)
- Varying is the default
- Each SIMD lane gets a unique value.
- Width dependent on target.

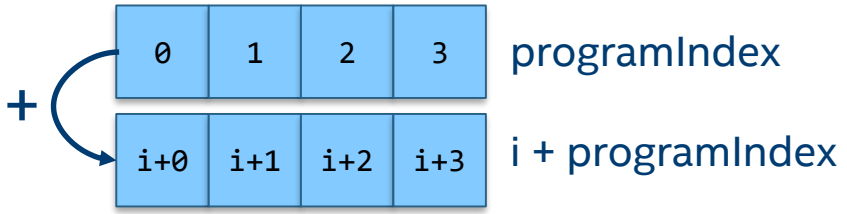


Built in variables

- `programCount`
  - Has type ‘`uniform int`’
  - Returns the vector width used in the compilation unit, 4 (SSE), 8 (AVX)
  - In this case the number of 32bit values that are packed into a vector register
  - The width of a varying variable

- `programIndex`
  - Has type ‘`varying int`’
  - Initialized to `{0 … programCount-1}`

  - Useful for indexing into arrays from a uniform base.



Control Flow

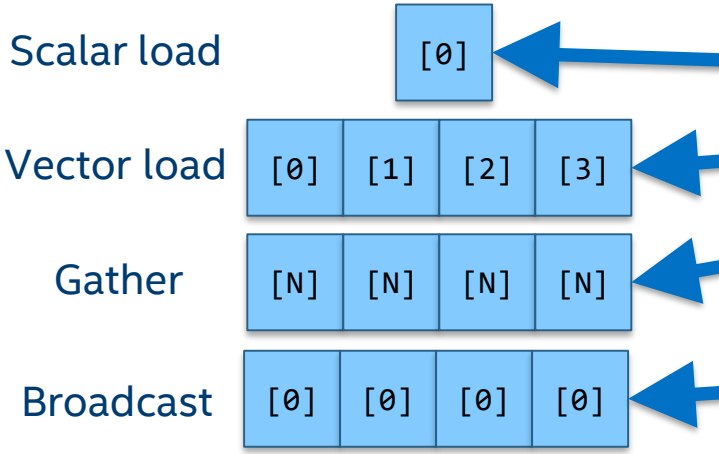
- ISPC has all the control flow constructs you'd find in C/C++
  - Conditionals
    - `if, else, switch`
  - Loops
    - `for, while, do…while`
- It also adds several new ones for convenience and performance
  - `foreach, foreach_active, foreach_tiled, foreach_unique`
  - `cif, cwhile, cdo, cfor`

Control Flow

- Special iteration constructs
  - foreach(i = 0 … N, [j = A … B, k = C … D])
    - Iterate over the range 0 … N in chunks of programCount elements
    - Vars will have type varying int
    - Remainder iterations properly masked
  - foreach_active(i)
    - Serially iterate over the active lanes
  - foreach_unique(val in x)
    - Serially iterate over unique values in varying value x

Arrays

- Arrays work as expected
  - Arrays of uniforms are just like C/C++ arrays. Can be passed from C/C++.



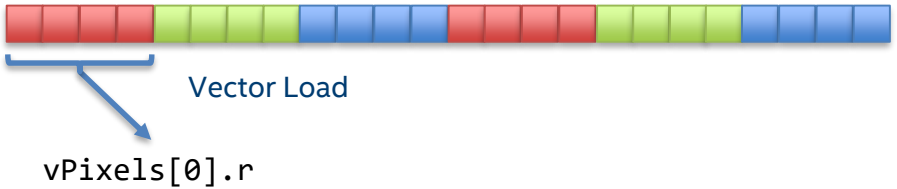

```
// array of 100 uniform floats
uniform float stuff[100];

// array of 100 varying floats
// => 100 * programCount, or 400+ total floats
varying float moreStuff[100];

void func1(uniform int input1[]) {
  uniform int foo = input1[0];

  varying int bar = input1[programIndex];

  varying int baz = input1[bar];
}

void func2(uniform int input2[]) {
  varying int foo = input2[0];
}
```

Structures

```
struct Color {
  float r, g, b;
};

uniform Color uPixels[100];
… = uPixels[programIndex].r;

varying Color vPixels[25];
… = vPixels[0].r;
```

```
struct cpp_varying_Color {
  float r[VLEN];
  float g[VLEN];
  float b[VLEN];
};
```

Memory Representation



Packed vector loads are more performant where possible!

Example ASM Output

ISPC Code

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

Emitted ASM

```
…
 vbroadcastss    .LCPI1_0(%rip), %ymm0
 vbroadcastss    .LCPI1_1(%rip), %ymm1
 vbroadcastss    .LCPI1_2(%rip), %ymm2
…
 vmovups         (%rdi,%rax), %ymm3
 vmulps          (%rsi,%rax), %ymm1, %ymm4
 vfmadd213ps      %ymm4, %ymm0, %ymm3
 vmovups         (%rdx,%rax), %ymm4
 vfmadd213ps      %ymm3, %ymm2, %ymm4
 vmovups          %ymm4, (%rcx,%rax)
…
```

- 3 loads, 3 multiplies, 2 adds and 1 store

Memory & Performance

- ISPC is awesome at generating the code for you but it can't rearrange your data and it can't speed up memory accesses for you
  - Data layout is important
  - Data needs to be in cache and it needs to be in the right layout
  - gather/scatter instructions can be painful
  - Prefer SoA, or AoSoA memory layouts, these will generate vector loads/stores
- Mike Acton, Data Oriented Design and C++
  - Check out Mike's talk from CppCon 2014

Language Features

- ISPC provides a rich stdlib of operations:
  - Logical operators
  - Bit ops
  - Math
  - Clamping and Saturated Arithmetic
  - Transcendental Operations
  - RNG (Not the fastest!)
  - Mask/Cross-lane Operations
  - Reductions
  - And that's not all!
- ISPC provides other features not covered here:
  - Pointers and Memory Allocations
  - AoS to SoA Helper Functions
  - C++ Like References
  - Binary Operator Overloading for Structures (+, -, *, /, <<, >>)
  - Built-in Task System
  - Multiple Math Libraries (Standard, Fast, SVML, System)

Why is this good?

- Programmers no longer need to know the ISA to write good vector code.
  - More accessible to programmers who aren't familiar with SIMD intrinsics.
  - More programmers able to fully utilize the CPU in various areas of game development.
- It's easier to read and maintain. It looks like scalar code.
- Supporting a new ISA is as easy as changing a command line option and recompiling.
- It's common to achieve big speedups on 4-wide SSE units and even bigger on CPUs with 8-wide AVX2 units without the difficulty of writing intrinsics.

Putting it all together: Dot4

Calculate Dot4 product of a bunch of vectors in a loop

Uses 4-wide XMM registers

```
void DotProductIntrinsic1(
    ispc::FVector4 *Result,
    const ispc::FVector4 *V1,
    const ispc::FVector4 *V2,
    const int Instances)
{
    for (int i = 0; i < Instances; i++)
    {
        const __m128 S1 = _mm_load_ps(&V1[i].V.v[0]);
        const __m128 S2 = _mm_load_ps(&V2[i].V.v[0]);

        __m128 Temp1, Temp2;
        Temp1 = _mm_mul_ps(S1, S2);
        Temp2 = _mm_shuffle_ps( Temp1, Temp1, SHUFFLEMASK(2,3,0,1) );
        Temp1 = _mm_add_ps(Temp1, Temp2);
        Temp2 = _mm_shuffle_ps( Temp1, Temp1, SHUFFLEMASK(1,2,3,0) );

        _mm_store_ps(&Result[i].V.v[0], _mm_add_ps(Temp1, Temp2));
    }
}
```

Putting it all together: Dot4

More speed needed! Use 256bit registers.

Usual solution: Write more intrinsics and guard for platforms that don't have AVX.

Two dot4s in the top loop and one in the bottom to catch the odd cases.

```
void DotProductIntrinsic2(
    ispc::FVector4* Result,
    const ispc::FVector4* V1,
    const ispc::FVector4* V2,
    const int Instances)
{
    const int InstancesBase = Instances & ~(1);

    for (int i = 0; i < InstancesBase; i+=2)
    {
        const __m256 S1 = _mm256_loadu_ps(&V1[i].V.v[0]);
        const __m256 S2 = _mm256_loadu_ps(&V2[i].V.v[0]);

        __m256 Temp1, Temp2;
        Temp1 = _mm256_mul_ps(S1, S2);
        Temp2 = _mm256_shuffle_ps(Temp1, Temp1, SHUFFLEMASK(2, 3, 0, 1));
        Temp1 = _mm256_add_ps(Temp1, Temp2);
        Temp2 = _mm256_shuffle_ps(Temp1, Temp1, SHUFFLEMASK(1, 2, 3, 0));

        _mm256_storeu_ps(&Result[i].V.v[0], _mm256_add_ps(Temp1, Temp2));
    }

    for (int i = InstancesBase; i < Instances; i++)
    {
        const __m128 S1 = _mm_load_ps(&V1[i].V.v[0]);
        const __m128 S2 = _mm_load_ps(&V2[i].V.v[0]);

        __m128 Temp1, Temp2;
        Temp1 = _mm_mul_ps(S1, S2);
        Temp2 = _mm_shuffle_ps(Temp1, Temp1, SHUFFLEMASK(2, 3, 0, 1));
        Temp1 = _mm_add_ps(Temp1, Temp2);
        Temp2 = _mm_shuffle_ps(Temp1, Temp1, SHUFFLEMASK(1, 2, 3, 0));

        _mm_store_ps(&Result[i].V.v[0], _mm_add_ps(Temp1, Temp2));
    }
}
```

Dot4: ISPC Uniform

Same as first example. Uses 4-wide XMM registers.

Uniform loop counter means iterate that many times through the loop.

```
struct FVector4
{
    float<4> V;
};

inline uniform FVector4 VectorDot4(
    const uniform FVector4& Vec1,
    const uniform FVector4& Vec2 )
{
    uniform FVector4 Temp1, Temp2;
    Temp1 = Vec1 * Vec2;
    Temp2 = VectorSwizzle(Temp1, 2,3,0,1);
    Temp1 = Temp1 + Temp2;
    Temp2 = VectorSwizzle(Temp1, 1,2,3,0);
    return Temp1 + Temp2;
}

export void DotProductUniform(uniform FVector4 Result[],
                              const uniform FVector4 Source1[],
                              const uniform FVector4 Source2[],
                              const uniform int Instances)
{
    for(uniform unsigned int i = 0; i < Instances; i++)
    {
        Result[i] = VectorDot4(Source1[i], Source2[i]);
    }
}
```

Dot4: ISPC Varying

Improvement! Now doing 4, 8 or 16 dot4s in parallel, one per SIMD lane depending on instruction set available.

Foreach handles cases where loop counter doesn't match up with SIMD width.

Still a problem. Remember gather/scatter? Data is still in AOS layout and won't use vector loads/stores.

```
inline FVector4 VectorDot4(const FVector4& Vec1, const FVector4& Vec2)
{
    FVector4 Temp1, Temp2;
    Temp1 = Vec1 * Vec2;
    Temp2 = VectorSwizzle(Temp1, 2,3,0,1);
    Temp1 = Temp1 + Temp2;
    Temp2 = VectorSwizzle(Temp1, 1,2,3,0);
    return Temp1 + Temp2;
}

export void DotProductVarying(uniform FVector4 Result[],
                              const uniform FVector4 Source1[],
                              const uniform FVector4 Source2[],
                              const uniform int Instances)
{
    foreach(i = 0 ... Instances)
    {
        const FVector4 S1 = Source1[i];
        const FVector4 S2 = Source2[i];

        Result[i] = VectorDot4(S1, S2);
    }
}
```

Dot4: ISPC Exotic

What if you combined best of intrinsics and varying?

Auto scaling: Now doing 1, 2 or 4 dot4s per iteration depending on SSE / AVX / AVX512.

With unrolling, this can be very effective. ISPC coalesces vector loads/stores.

Useful for certain algorithms, like normalization.

```
struct WideFVector4
{
    float V[programCount];
};

unmasked inline void LoadWideFVector4(
    uniform FVector4 * uniform DstPtr,
    const uniform FVector4* uniform SrcPtr)
{
    *DstPtr = *SrcPtr;
#if TARGET_WIDTH == 8 || TARGET_WIDTH == 16
    *(DstPtr + 1) = *(SrcPtr + SrcStride);
#endif
#if TARGET_WIDTH == 16
    *(DstPtr + 2) = *(SrcPtr + 2*SrcStride);
    *(DstPtr + 3) = *(SrcPtr + 3*SrcStride);
#endif
}

static inline uniform WideFVector4 WideVectorDot4(
    const uniform WideFVector4& Vec1,
    const uniform WideFVector4& Vec2 )
{
    uniform WideFVector4 Temp1, Temp2;
    Temp1 = Vec1 * Vec2;
    Temp2 = VectorSwizzle(Temp1, 2,3,0,1);
    Temp1 = Temp1 + Temp2;
    Temp2 = VectorSwizzle(Temp1, 1,2,3,0);
    return Temp1 + Temp2;
}
```

```
export void DotProductWide(uniform FVector4 Result[],
                           const uniform FVector4 Source1[],
                           const uniform FVector4 Source2[],
                           const uniform int Instances)
{
    uniform int InstancesBase = Instances & ~(programCount-1);
    uniform int InstancesOffset = programCount / 4;

    for(uniform int i = 0; i < InstancesBase; i+=InstancesOffset)
    {
        uniform WideFVector4 S1, S2;

        LoadWideFVector4((uniform FVector4 *uniform)&S1,
                         (uniform FVector4 *uniform)&Source1[i]);
        LoadWideFVector4((uniform FVector4 *uniform)&S2,
                         (uniform FVector4 *uniform)&Source2[i]);

        const uniform WideFVector4 DotResult = WideVectorDot4(S1, S2);

        StoreWideFVector4((uniform FVector4 *uniform)&Result[i],
                         (uniform FVector4 *uniform)&DotResult);
    }

    for(uniform unsigned int i = InstancesBase; i < Instances; i++)
    {
        Result[i] = VectorDot4(Source1[i], Source2[i]);
    }
}
```

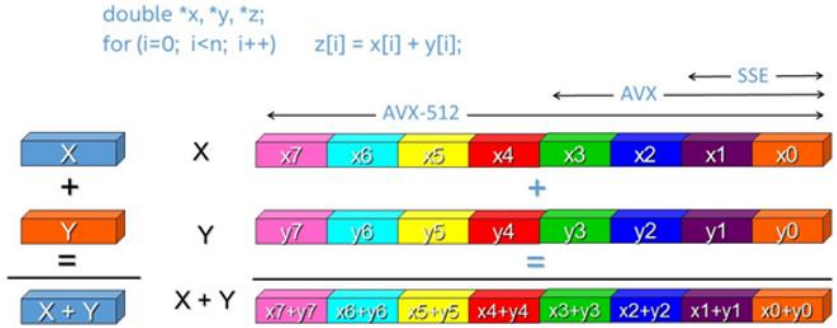
UNREAL AND ISPC

Unreal ISPC Integration

ISPC available in Unreal from 4.23. Console support added in 4.25.

Used in Chaos physics and animation systems! Supports custom usage.

- Include ISPC module in your build.cs
- Add ispc files to your project
- Include a generated C++ header
- Unreal build tool handles the rest



Scalar and vectorized loop versions with Intel® SSE, AVX and AVX-512.

When to use ISPC in Unreal?

- Good for dense compute-bound workloads. Heavy math like physics intersection testing, cloth or CPU vertex transformations
- Best with contiguous memory load, manipulate, store ie Unreal TArray
- Best when no data dependencies between operations. Especially useful when combined with ParallelFor and batching

CHAOS AND ISPC

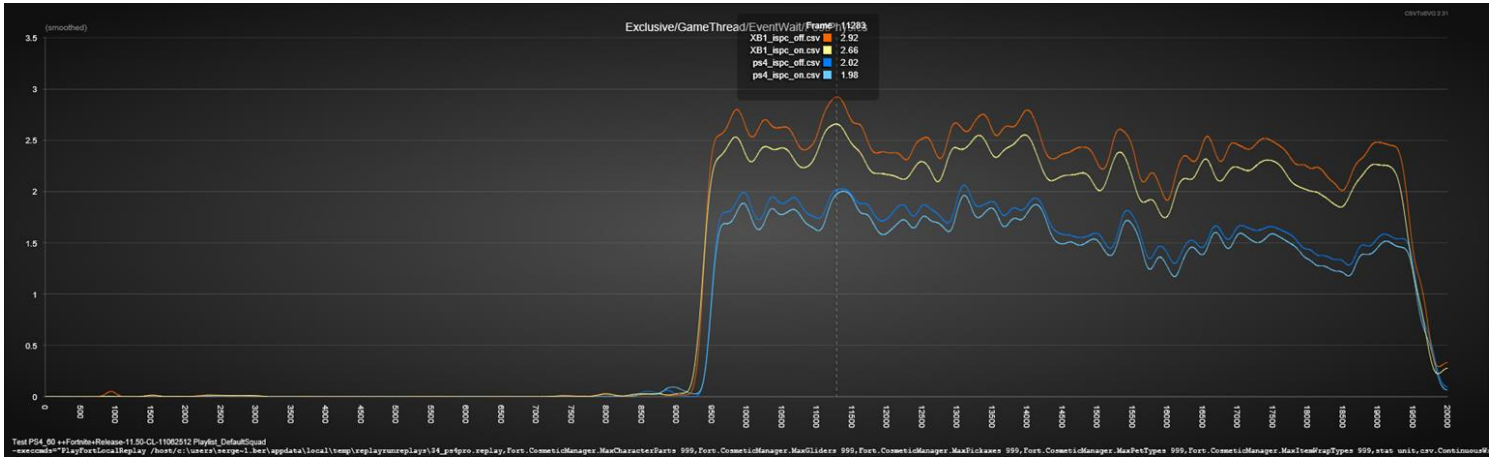
What is Chaos?

New physics engine inside UE4

- Non Convex Collisions
- Fields
- Niagara Integration
- Interactive Caching
- Dedicated Physics Thread
- Geometry Collection
- Cutting Tools
- Destructible LODs
- Dynamic Strain Evaluation

Benefits of ISPC for Chaos

- ISPC provides a simple shader language like interface for performance optimizations using SIMD
- Works across platforms avoiding the need for platform specific intrinsic code
- Actively used in by Chaos in both Fortnite and Destruction



Rigid Skinning

Generate transformed vertices with an input vertex buffer and transform dictated by an array of bones.

Bones often repeat, use foreach_unique to cut down the number of transform ops needed

Short vector arrays are packed, use aos_to_soa to eliminate gathers

```
export void SetDynamicData_RenderThread(uniform FVector PositionBuffer[],
                                        const uniform unsigned int NumVertices,
                                        const uniform unsigned int Stride,
                                        const uniform int32 BoneMap[],
                                        const uniform FMatrix Transforms[],
                                        const uniform FVector Vertices[])
{
    uniform unsigned int Chunk = 0;

    foreach(i = 0 ... NumVertices)
    {
        uniform float * uniform pVertices = (uniform float * uniform)&Vertices[Chunk];
        uniform float * uniform pPosition = (uniform float * uniform)&PositionBuffer[Chunk];

        FVector P;
        aos_to_soa3(pVertices, &P.V[0], &P.V[1], &P.V[2]);

        FVector Out;
        const int32 Bone = BoneMap[i];
        foreach_unique(Index in Bone)
        {
            Out = VectorTransformVector(P, Transforms[Index]);
        }

        soa_to_aos3(Out.V[0], Out.V[1], Out.V[2], pPosition);

        Chunk += programCount;
    }
}
```

Bounding Box

Custom reduce

Problem is that with large arrays of boxes to sum, occasionally the sum will be negative

Doing a reduce_min and reduce_max seems right what if a box is invalid? Then you're clamping to zero (default init) instead of the value you want

Foreach_active and operator+ serializes and handles this case

```
export void BoxCalcBounds(const uniform int8 * uniform BoneHierarchy,
                          const uniform int BoneNodeStatusFlagsOffset,
                          const uniform int BoneNodeStride,
                          const uniform int TransformIndices[],
                          const uniform FMatrix GlobalMatrices[],
                          const uniform FBox BoundingBoxes[],
                          const uniform FMatrix &LocalToWorldWithScale,
                          uniform FBox &BoundingBox,
                          const uniform int NumBoxes)
{
    FBox BoundingBoxSum = BoxDefaultInit();

    foreach(BoxIdx = 0 ... NumBoxes)
    {
        const int TransformIndex = TransformIndices[BoxIdx];

        const int Offset = BoneNodeStride * BoxIdx + BoneNodeStatusFlagsOffset;
        #pragma ignore warning(perf)
        const unsigned int BoneNodeStatusFlags = (unsigned int)BoneHierarchy[Offset];

        #pragma ignore warning(perf)
        const FBox B = BoundingBoxes[BoxIdx];

        if (IsGeometry(BoneNodeStatusFlags) && B.IsValid)
        {
            #pragma ignore warning(perf)
            const FMatrix M = GlobalMatrices[TransformIndex];
            BoundingBoxSum = BoundingBoxSum + BoxTransformByMatrix(B, M * LocalToWorldWithScale);
        }
    }

    // Custom reduce for large arrays
    uniform FBox ReducedBox[programCount];
    #pragma ignore warning(perf)
    ReducedBox[programIndex] = BoundingBoxSum;

    foreach_active(j)
    {
        unmasked
        {
            BoundingBox = BoundingBox + ReducedBox[j];
        }
    }
}
```

Scene Queries

Process multiple elements of a list simultaneously

Can directly call cpp code from ispc

Performance can vary depending on the size of the list. Scales well but does have overhead so ymmv but more elements does better.

```
extern "C" unmasked void QueryIntersectsHelperCPP(const uniform TPayloadBoundsElement* uniform Elem,
    void * uniform VisitorPtr, uniform FQueryFastData& CurData, const uniform EAABBQueryType Query,
    const uniform bool bUseInstanceBounds, uniform bool &bContinue);

export uniform bool QueryIntersectsRaycast(
    const uniform TPayloadBoundsElement Elements[], const uniform FVector& Start,
    uniform FQueryFastData& CurData, uniform float& TOI, uniform FVector& TmpPosition,
    const uniform FCollisionFilterData * uniform QueryFilterData, void * uniform VisitorPtr,
    const uniform bool bUseInstanceBounds, const uniform uint32 NumElements)
{
    uniform uint32 QuerierBit;
    const uniform bool bShouldPrePreFilter = ShouldPrePreFilter(QueryFilterData, QuerierBit);
    uniform bool bContinue = true;

    foreach (i = 0 ... NumElements)
    {
        if(bContinue == true)
        {
            const uint32 Word1 = Elements[i].Payload.UnionFilterData.Word1;
            const uint32 Word2 = Elements[i].Payload.UnionFilterData.Word2;
            const bool bCanPrePreFilter = (Elements[i].Payload.bCanPrePreFilter > 0) ? true : false;

            if (!PrePreFilter(bShouldPrePreFilter, bCanPrePreFilter, QuerierBit, Word1, Word2))
            {
                const AABB Bounds = LoadAABB(Elements, i);
                if(RaycastAABB(Bounds, Start, CurData.Dir, CurData.InvDir, CurData.bParallel,
                    CurData.CurrentLength, CurData.InvCurrentLength, TOI, TmpPosition) > 0)
                {
                    foreach_active(j)
                    {
                        unmasked
                        {
                            if(bContinue == true)
                            {
                                const uniform int Index = extract(i, j);
                                QueryIntersectsHelperCPP(&Elements[Index], VisitorPtr, CurData,
                                    Raycast, bUseInstanceBounds, bContinue);
                            }
                        }
                    }
                }
            }
        }
    }
    return bContinue;
}
```

Rigid Chains

Simple math heavy calculations

Convert all data to uniforms so it performs optimally

No cost compared to native intrinsics

Overall 15-20% faster for a character skin

```
export void ApplyPositionProjection(
    uniform FJointSolverGaussSeidel * uniform M,
    const uniform FVector &CX,
    const uniform float CXLen,
    const uniform float ParentMassScale,
    const uniform float Stiffness)
{
    const uniform FVector CXDir = CX / CXLen;
    const uniform FVector V0 = M->Vs[0] + VectorCross(M->Ws[0], M->Xs[0] - M->Ps[0]);
    const uniform FVector V1 = M->Vs[1] + VectorCross(M->Ws[1], M->Xs[1] - M->Ps[1]);
    const uniform FVector CV = VectorDot(V1 - V0, CXDir) * CXDir;

    const uniform float IM0 = ParentMassScale * M->InvMs[0];
    const uniform float IM1 = M->InvMs[1];
    const uniform FVector IIL0 = ParentMassScale * M->InvILs[0];
    const uniform FVector& IIL1 = M->InvILs[1];
    uniform FMatrix33 J0 = Matrix33Zero;
    if(IM0 > 0)
    {
        J0 = ComputeJointFactorMatrix(M->Xs[0] - M->Ps[0], M->InvIs[0], IM0);
    }
    const uniform FMatrix33 J1 = ComputeJointFactorMatrix(M->Xs[1] - M->Ps[1], M->InvIs[1], IM1);
    const uniform FMatrix33 IJ = MatrixInverse(AddAB(J0, J1));

    const uniform FVector DX = Multiply(IJ, CX);
    const uniform FVector DV = Multiply(IJ, CV);

    const uniform FVector DP0 = IM0 * DX;
    const uniform FVector DP1 = -IM1 * DX;
    const uniform FVector DR0 = Multiply(M->InvIs[0], VectorCross(M->Xs[0] - M->Ps[0], DX));
    const uniform FVector DR1 = Multiply(M->InvIs[1], VectorCross(M->Xs[1] - M->Ps[1], VectorNegate(DX)));

    const uniform FVector DV0 = IM0 * DV;
    const uniform FVector DV1 = -IM1 * DV;
    const uniform FVector DW0 = Multiply(M->InvIs[0], VectorCross(M->Xs[0] - M->Ps[0], DV));
    const uniform FVector DW1 = Multiply(M->InvIs[1], VectorCross(M->Xs[1] - M->Ps[1], VectorNegate(DV)));

    ApplyPositionDelta(M, Stiffness, DP0, DP1);
    ApplyRotationDelta(M, Stiffness, DR0, DR1);
    ApplyVelocityDelta(M, Stiffness, DV0, DW0, DV1, DW1);
}
```

ANIMATION AND ISPC

Animation in UE4

Unreal Engine supports 8 platforms currently, 10 with next gen consoles

Maintaining SIMD code across all targets is a large undertaking that the animation team historically has not had the time to commit to

Performance improvements have focused instead on high level algorithmic changes or multithreading

Previously the only vectorization in animation came from generic library code in UE4.

Animation in Fortnite

Fortnite is driving character performance improvements across the engine

Large number of characters (100 in BR)

LTM's such as Team Rumble/50v50 tend towards big groups of players close together, hampering usual performance 'tricks'

Performance demands on animation always increasing with new gameplay (e.g. NPCs)

Blue sky? If we could run 2x as many characters what new gameplay would that unlock

ISPC

Performance is always critical for animation (more complicated characters, more characters on screen at once etc).

Ability to write code once and hit all target platforms is a big win for us.

Lets team focus on building animation tech instead instead of maintaining N versions of the same logic.

Uses in UE4 Animation

Focus on runtime hotspots first:

- Pose Blending
- Additive Pose Conversion
- Normalize Rotations
- Decompressing Animation Data
- Building Component Space Transforms
- Preparing Bone Transforms for Renderer

Pose Blending

Ideal scenario for ISPC

Given 2 bone transform arrays and a weight create a 3rd transform array

Previous optimization work meant we were already running through contiguous transform arrays

In testing we saw a near 2x performance improvement from ISPC

```
export void BlendTransformOverwrite(const uniform FTransform SourcePose[],
                                    uniform FTransform ResultPose[],
                                    const uniform float BlendWeight,
                                    const uniform int NumBones)
{
    for(uniform int BoneIndex = 0; BoneIndex < NumBones; BoneIndex++)
    {
        ResultPose[BoneIndex] = SourcePose[BoneIndex] * BlendWeight;
    }
}

export void BlendTransformAccumulate(const uniform FTransform SourcePose[],
                                    uniform FTransform ResultPose[],
                                    const uniform float BlendWeight,
                                    const uniform int NumBones)
{
    for(uniform int BoneIndex = 0; BoneIndex < NumBones; BoneIndex++)
    {
        const uniform FTransform Source = SourcePose[BoneIndex];
        uniform FTransform Dest = ResultPose[BoneIndex];

        const uniform FVector4 BlendedRotation = Source.Rotation * BlendWeight;
        Dest.Rotation = VectorAccumulateQuaternionShortestPath(Dest.Rotation, BlendedRotation);

        Dest.Translation = VectorMultiplyAdd(Source.Translation, BlendWeight, Dest.Translation);
        Dest.Scale3D = VectorMultiplyAdd(Source.Scale3D, BlendWeight, Dest.Scale3D);

        ResultPose[BoneIndex] = Dest;
    }
}
```

Decompression

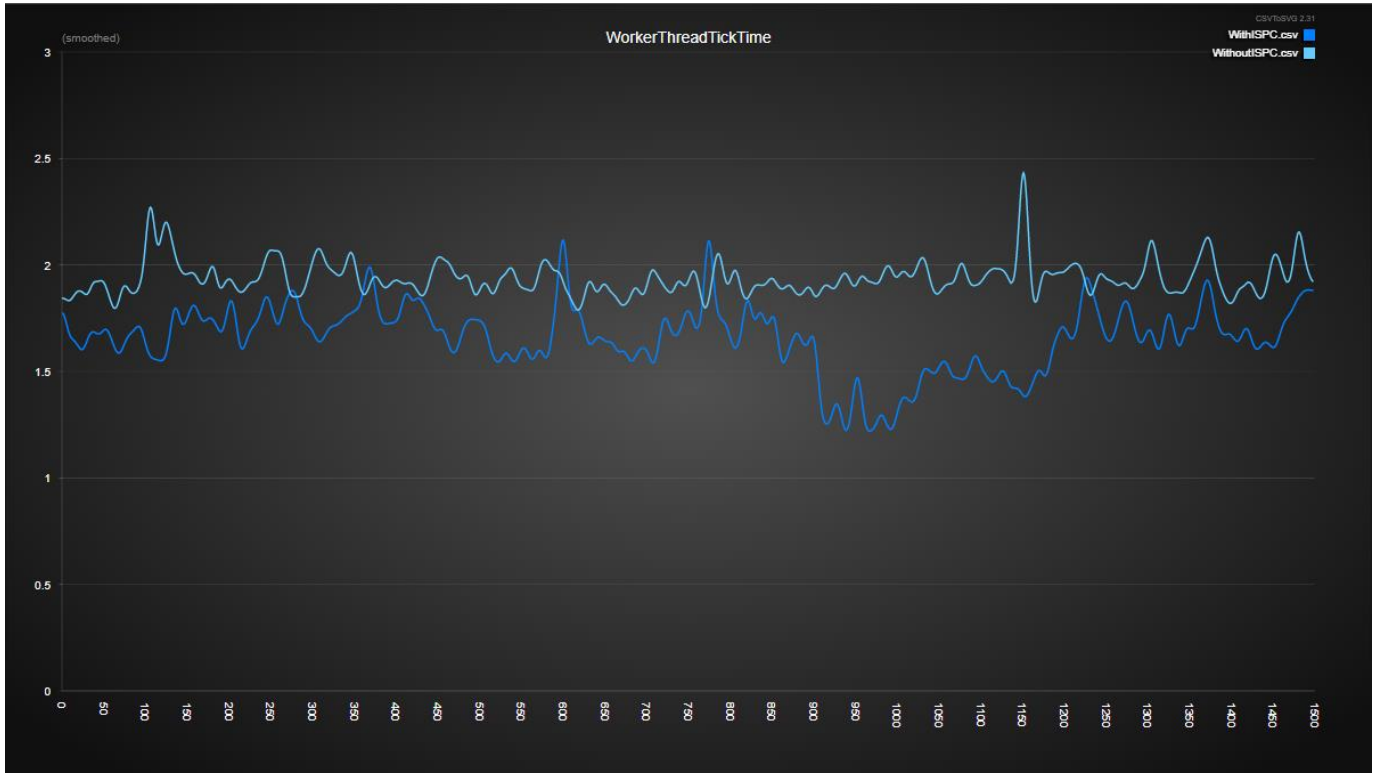
Decompression is a big part of the performance cost of animation in UE4

Each supported compression format in the engine has ISPC code written for it

In testing we we saw between 1.5x and 2x performance improves depending on the type of compression used

```
foreach(PairIndex = 0 ... PairCount)
{
    #pragma ignore warning(perf)
    const BoneTrackPair Pair = DesiredPairs[PairIndex];
    const int TrackIndex = Pair.TrackIndex;
    const int AtomIndex = Pair.AtomIndex;

    // call the decoder directly (not through the vtable)
    const int* TrackData = CompressedTrackOffsets + (TrackIndex * 4);

    #pragma ignore warning(perf)
    int RotKeysOffset = TrackData[2];

    #pragma ignore warning(perf)
    int NumRotKeys = TrackData[3];
    const unsigned int8* RotStream = CompressedByteStream + RotKeysOffset;

    FVector4 R0;

    if (NumRotKeys == 1)
    {
        // For a rotation track of n=1 keys, the single key is packed as an FQuatFloat96NoW.
        DecompressRotation(R0, RotStream, RotStream, ACF_Float96NoW);
    }
    else
    {
        int Index0;
        int Index1;
        const float Alpha = TimeToIndex(SequenceLength, RelativePos, NumRotKeys, InterpolationType, Index0, Index1);

        // unpack a single key first
        const unsigned int8* KeyData0= RotStream + RotationStreamOffset +(Index0*CompressedRotationStrideNum);
        DecompressRotation(R0, RotStream, KeyData0, FORMAT);

        if (Index0 != Index1)
        {
            // unpack and lerp between the two nearest keys
            const unsigned int8* KeyData1= RotStream + RotationStreamOffset +(Index1*CompressedRotationStrideNum);
            FVector4 R1;

            DecompressRotation(R1, RotStream, KeyData1, FORMAT);

            // Fast linear quaternion interpolation.
            const FVector4 BlendedQuat = QuatFastLerp(R0, R1, Alpha);
            R0 = VectorNormalizeQuaternion(BlendedQuat);
        }
    }
```

Performance



Downsides to ISPC?

Existing animation systems need rewriting to work with ISPC (rearrange data, remove branching and random access), existing code not DOD friendly.

Duplication of logic between ISPC and ‘vanilla’ code paths

Future Work

Animation Compression

Curve Blending

URO Interpolation

Retargeting

Wrap Up

- ISPC optimizations regularly show perf gains over C++. Plenty more not covered here!
- Unreal devs can use ISPC in their games from 4.23
- Full platform support in 4.25 (Win, Mac, Linux, PS4, Xbox, ARM).

Feedback welcome! Twitter handle **@jeff_rous**

Links

ISPC Project (ispc.github.io)

Causing Chaos: The Future of Physics and Destruction in Unreal Engine (youtube.com/watch?v=6T8LzaIq3Qs)

GDC Optimization Talk (gdcvault.com/browse/gdc-19/play/1026175)

CPU Particles (software.intel.com/en-us/articles/maximizing-visuals-with-cpu-particles-in-unreal-engine-4)

Optimization Guide (software.intel.com/en-us/articles/unreal-engine-4-optimization-tutorial-part-1)

CPU Optimizations for Cloth Simulations (software.intel.com/en-us/articles/unreal-engine-4-blueprint-cpu-optimizations-for-cloth-simulations)


UNREAL ENGINE 4
GET IT MEOW!


intel®
experience
what's inside™


UNREAL
ENGINE