EXPLOITING VECTORIZATION WITH ISPC

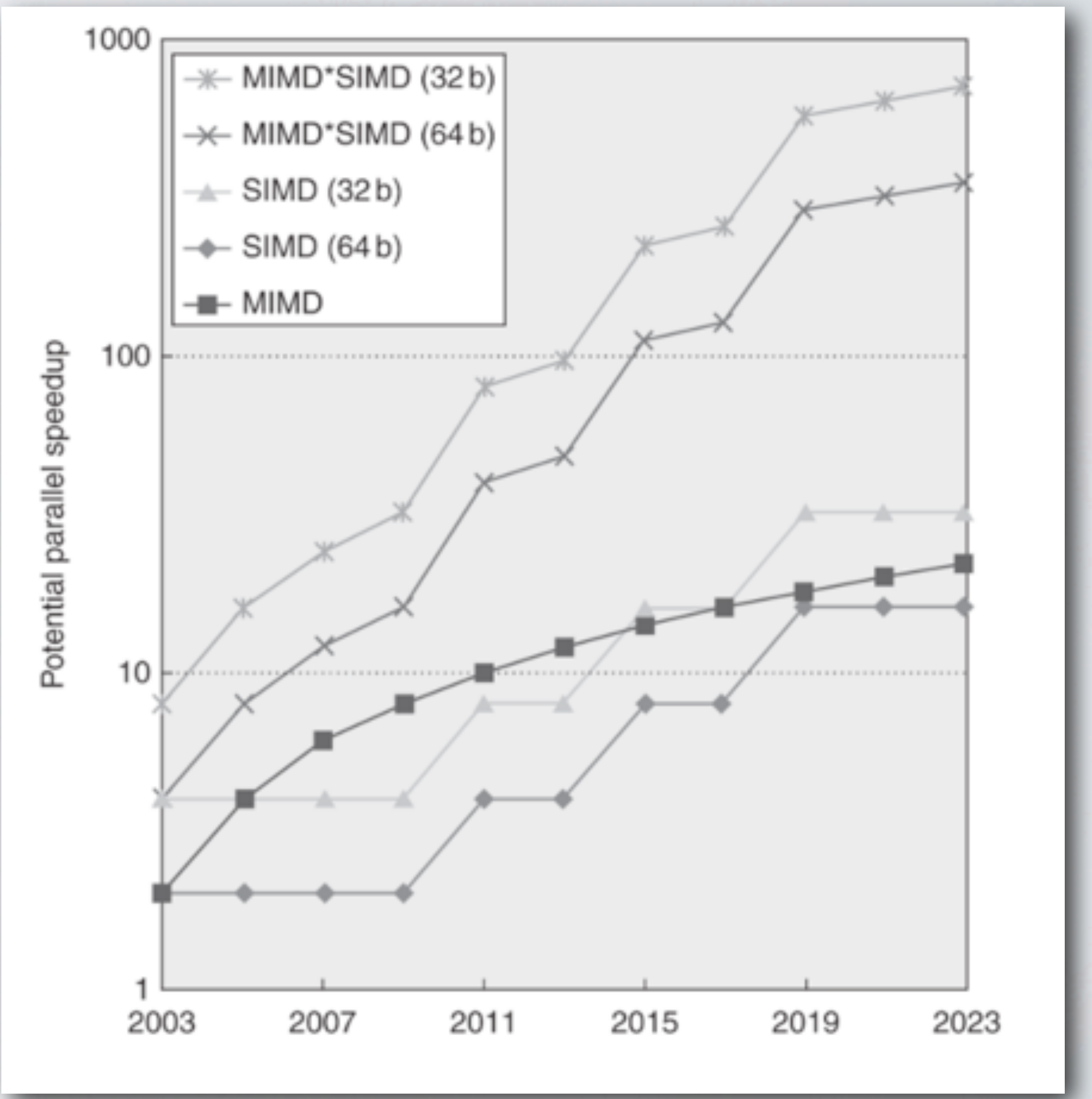
Roberto A. Vitillo (LBNL)

8/14/2013 Concurrency Forum Meeting

SIMD

WHY IT MATTERS

- **S**ingle **I**nstruction, **M**ultiple **D**ata:
  - processor throughput is increased by handling multiple data in parallel
  - exploiting SIMD is increasingly becoming more important on Xeons (see AVX-512)
  - exploiting SIMD is <u>mandatory</u> to achieve reasonable performances on the Xeon PHI



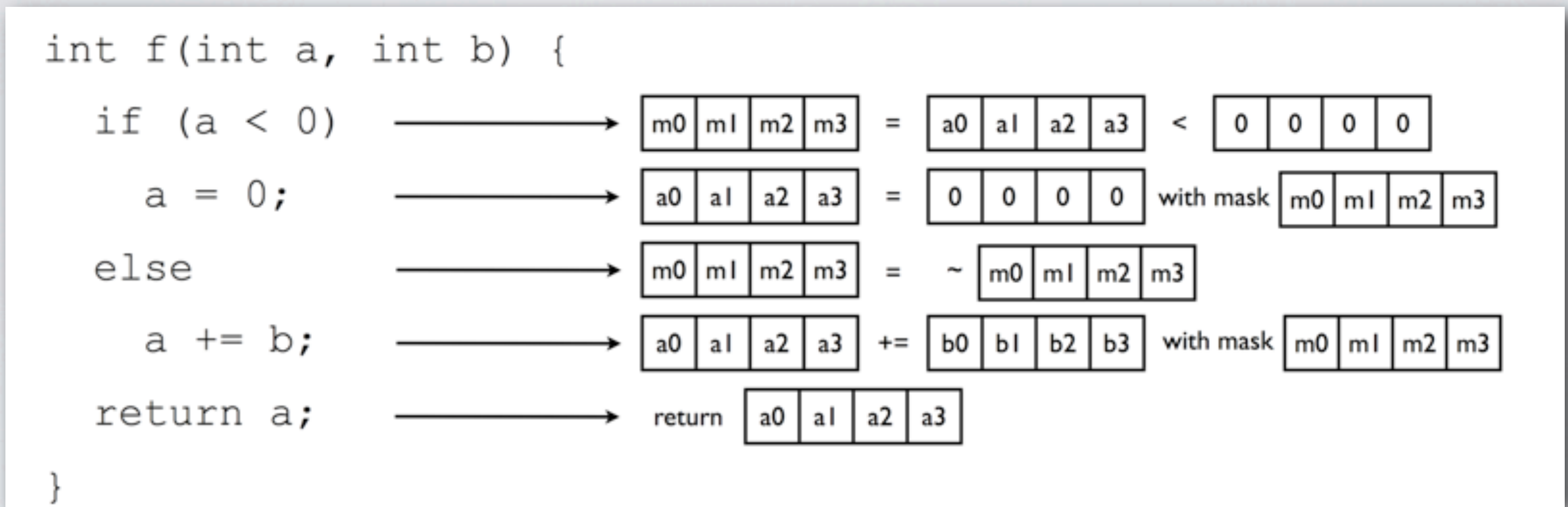
Source: "Computer Architecture, A Quantitative Approach"

MEET ISPC

- Intel SPMD Program Compiler (ISPC) extends a C-based language with “single program, multiple data” (SPMD) constructs
- An ISPC program describes the behavior of a single program instance
    - even though a “gang” of them is in reality being executed
    - gang size is usually no more than 2-4x the native SIMD width of the machine
- For CUDA affectionados
    - ISPC Program is similar to a CUDA thread
    - An ISPC gang is similar to a CUDA warp

SPMD PARADIGM

Execution of a SPMD program with a gang size of 4



- Observations:
  - diverging control flow reduces the utilization of vector instructions
  - vectorization adds masking overhead

HELLO WORLD

uniform variable is shared among program instances

make function available to be called from application code

foreach expresses a parallel computation

each program instance has a private instance of a non-uniform variable (a.k.a. varying variable)

```
export void simple(uniform float vin[], uniform float vout[],
                   uniform int count) {
    foreach (index = 0 ... count) {
        float v = vin[index];
        if (v < 3.)
            v = v * v;
        else
            v = sqrt(v);
        vout[index] = v;
    }
}
```

simple.ispc, compiled with ispc

```
#include <stdio.h>
#include "simple.h"

int main() {
    float vin[16], vout[16];
    for (int i = 0; i < 16; ++i)
        vin[i] = i;

    simple(vin, vout, 16);

    for (int i = 0; i < 16; ++i)
        printf("%d: simple(%f) = %f\n", i, vin[i], vout[i]);
}
```

main.c, compiled with GCC

ispc function is called like any other function from the C/C++ application

DEBUGGING SUPPORT

BEYOND GDB

```
foreach(k = 0 ... 6){
    int i = k * 7;

    print("%\n", i);

    double* dR   = &P[i];
    double* dA   = &P[i+3];

    ...
}
```

Prints [0, 7, 14, 21, 28, 35, ((42)), ((49))]

gang size of 8

0	7	14	21	28	35	(42)	(49)

Inactive Program Instances

DEBUGGING SUPPORT

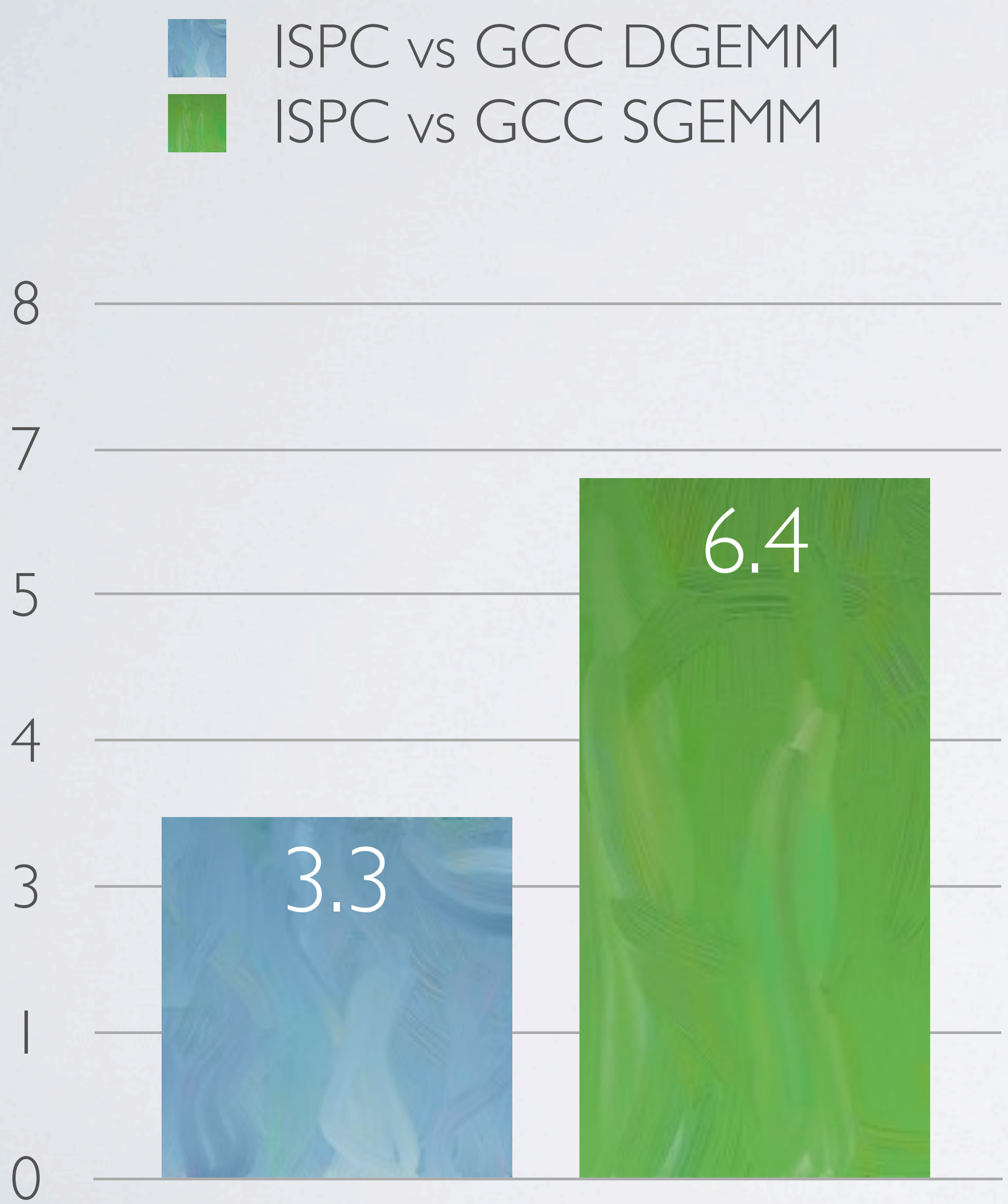
PERFORMANCE WARNINGS

```
export void foo(uniform float * uniform A, uniform int n){
    foreach(i = 0 ... n){
        A[i*8] *= A[i*8];
    }
}
```

```
vitillo@mickey /tmp $ ispc test.ispc -O2 -o test.o --target=avx
test.ispc:3:15: Performance Warning: Coalesced gather into 8
        loads (8 x 1-wide).
    A[i*8] *= A[i*8];
              ^^^^^^

test.ispc:3:5: Performance Warning: Scatter required to store
        value.
    A[i*8] *= A[i*8];
    ^^^^^^
```

MATRIX MULTIPLICATION

EXPLOITING HORIZONTAL VECTORIZATION WITH SMALL MATRICES




```
inline void mxm(uniform float * uniform A,
                uniform float * uniform B,
                uniform float * uniform C,
                uniform int M,
                uniform int N,
                uniform int K,
                uniform int nmat,
                int idx)
{
    for(uniform int i = 0; i < M; i++){
        for(uniform int j = 0; j < N; j++){
            float sum = 0;

            for(uniform int k = 0; k < K; k++){
                sum += A[i*K*nmat + k*nmat + idx] * B[k*N*nmat + j*nmat + idx];
            }

            C[i*N*nmat + j*nmat + idx] = sum;
        }
    }
}

export void gemm(uniform float * uniform A,
                 uniform float * uniform B,
                 uniform float * uniform C,
                 uniform int M,
                 uniform int N,
                 uniform int K,
                 uniform int nmat)
{
    foreach(i = 0 ... nmat){
        mxm(A, B, C, M, N, K, nmat, i);
    }
}
```

xGEMM 5x5 speedup over 1000 matrices (GCC 4.8 -O3, Ivy Bridge)

MATRIX MULTIPLICATION

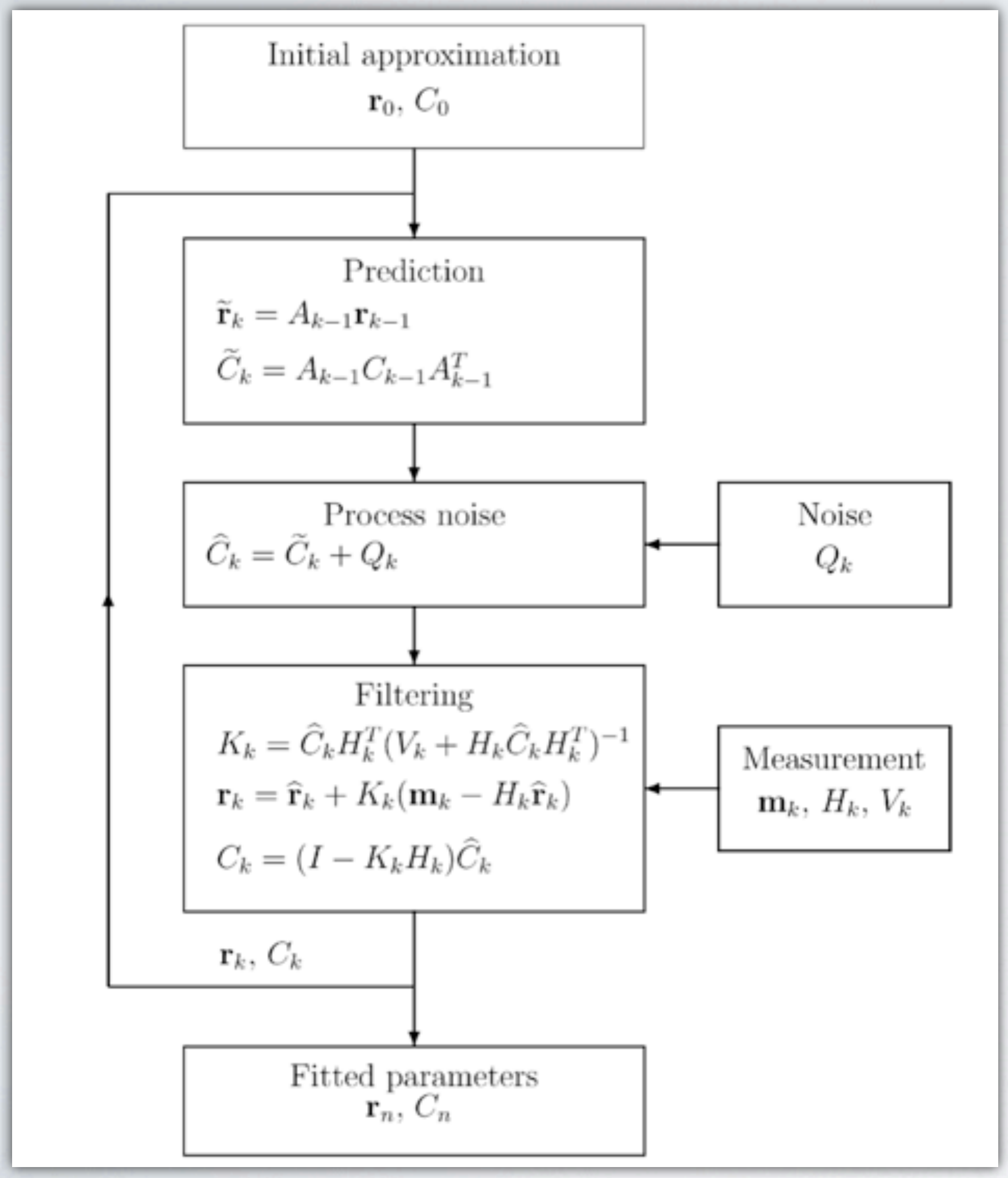
EXPLOITING HORIZONTAL VECTORIZATION WITH SMALL MATRICES


```
inline void mxm(uniform float * uniform A,
                uniform float * uniform B,
                uniform float * uniform C,
                uniform int M,
                uniform int N,
                uniform int K,
                uniform int nmat,
                int idx)
{
    for(uniform int i = 0; i < M; i++){
        for(uniform int j = 0; j < N; j++){
            float sum = 0;

            for(uniform int k = 0; k < K; k++){
                sum += A[i*K*nmat + k*nmat + idx] * B[k*N*nmat + j*nmat + idx];
            }

            C[i*N*nmat + j*nmat + idx] = sum;
        }
    }
}

export void gemm(uniform float * uniform A,
                 uniform float * uniform B,
                 uniform float * uniform C,
                 uniform int M,
                 uniform int N,
                 uniform int K,
                 uniform int nmat)
{
    foreach(i = 0 ... nmat){
        mxm(A, B, C, M, N, K, nmat, i);
    }
}
```

xGEMM 5x5 speedup over 1000 matrices (GCC 4.8 -O3, Ivy Bridge)

KALMAN FILTER

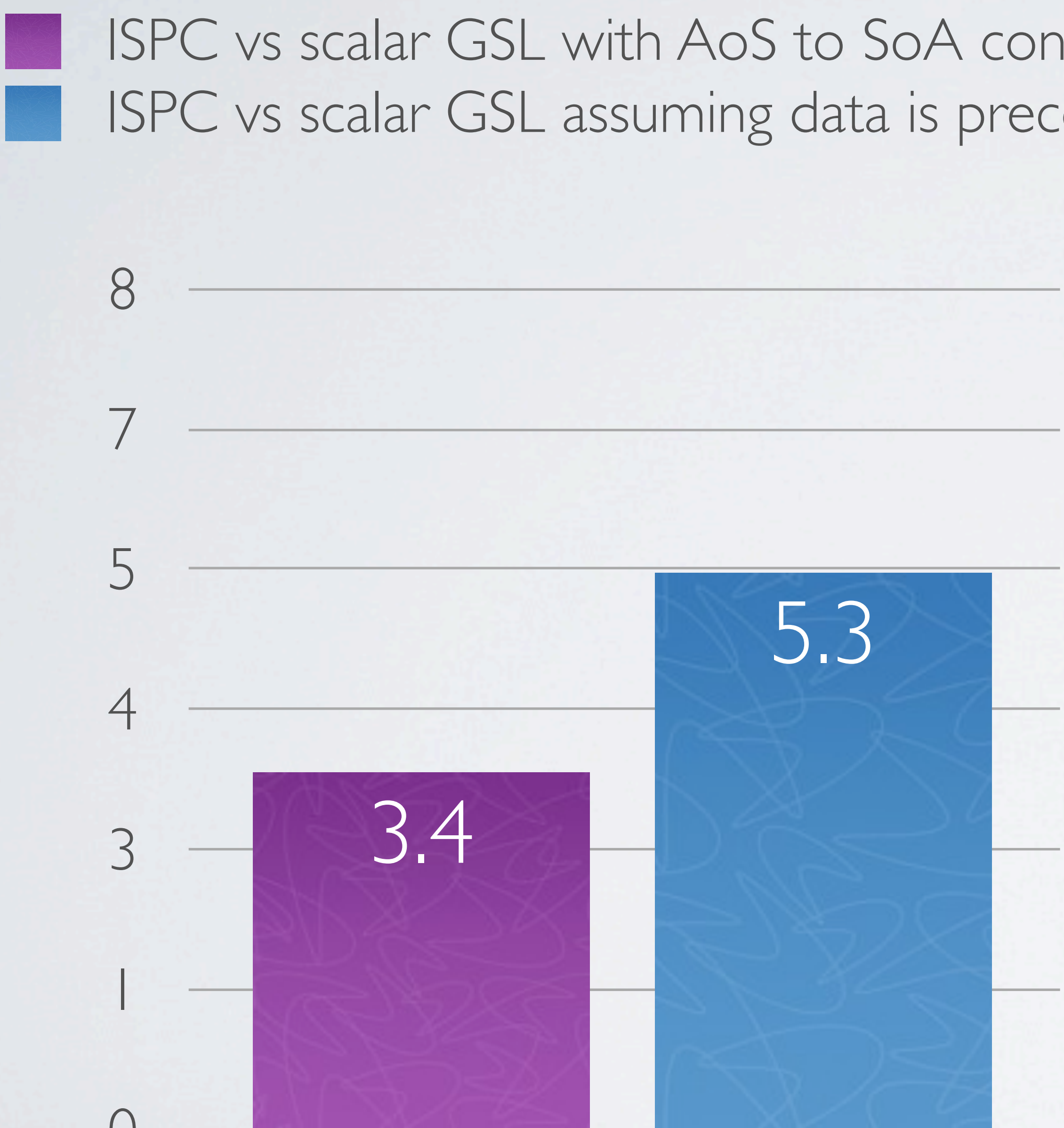
TRACKING



- The Kalman filter method is intended for finding the optimum estimation r of an unknown vector $\boldsymbol{r}^t$ according to the measurements m_k, k=1...n, of the vector $\boldsymbol{r}^t$.
- Plenty of linear algebra operations so it's a good use case for vectorization.
- Caveats:
  - tracks have different number of hits (use sorting)
  - an hit can be 1 or 2 dimensional (serialize branching)

KALMAN FILTER

100 EVENTS, ~100 TRACKS WITH ~10 HITS EACH



KalmanFilter speedup (double precision), Ivy Bridge

```
export void startFilter(uniform KalmanFilter * uniform filter,
                        uniform KalmanFilterParameter * uniform param){
    foreach(i = 0 ... filter->ntracks){
        filterTrack(filter, param, i);
    }
}

inline void filterTrack(uniform KalmanFilter * uniform filter,
                        uniform KalmanFilterParameter * uniform param,
                        int i){
    ...
    for(uniform int h = 0; h < param->max_hit_count; h++){
        if(h >= param->hit_count[i])
            continue;

        predictState(filter, param, h, i);
        predictCovariance(filter, param, h, i);

        if(param->hits[h].is2Dim[i]){
            ...
            correctGain2D(filter, i);
            correctState2D(filter, i);
            correctCovariance2D(filter, i);
        }else{
            ...
            correctGain1D(filter, i);
            correctState1D(filter, i);
            correctCovariance1D(filter, i);
        }
    }
   ...
}
```

WHAT ABOUT OPENCL?

FIRST IMPRESSIONS

- Intel's OpenCL embeds an implicit vectorization module which has some similarities with ISPC but...
  - ISPC warns the user if an inefficient data access pattern is detected
  - the programmer can specify in ISPC which code has to be executed serially and which one has to be vectorized (for vs foreach loop)
  - variables can be declared as uniform or varying in ISPC
  - ISPC supports lightweight kernel calls while in OpenCL an API call to a driver must be made
- OpenCL has native support for the Xeon PHI
  - ISPC will support the Xeon PHI natively when LLVM will
- Porting code from ISPC to OpenCL and vice versa is relatively easy
  - OpenCL comes with some boilerplate code though
  - task parallelism compositing with TBB works with ISPC and OpenCL
  - but ISPC is easier to compose with an arbitrary task scheduler while OpenCL requires device fission

WHAT ABOUT OPENCL?

FIRST IMPRESSIONS

```
__kernel void gemm(__global float *A,
                   __global float *B,
                   __global float *C,
                   const int M,
                   const int N,
                   const int K,
                   const int num){
    const int idx = get_global_id(0);

    if(idx >= num)
        return;

    for(int i = 0; i < M; i++){
        for(int j = 0; j < N; j++){
            float sum = 0;

            for(int k = 0; k < K; k++){
                sum += A[i*K*num + k*num + idx] * B[k*N*num + j*num + idx];
            }

            C[i*N*num + j*num + idx] = sum;
        }
    }
}
```

Why is it not using 256 bit registers?

AVX

```
.LBB0_8:
    movslq      %esi, %rsi
    vmovups     (%rdi,%rsi,4), %xmm2
    movslq      %ebp, %rbp
    vmovups     (%rcx,%rbp,4), %xmm3
    vmulps      %xmm2, %xmm3, %xmm2
    vaddps      %xmm2, %xmm1, %xmm1
    addl  %edx, %ebp
    addl  %eax, %esi
    decl  %r11d
    jne   .LBB0_8
```

AVX 2

```
.LBB0_8:
    movslq      %r11d, %r11
    vmovups     (%r14,%r11,4), %ymm2
    movslq      %esi, %rsi
    vmovups     (%r12,%rsi,4), %ymm3
    vmulps      %ymm2, %ymm3, %ymm2
    vaddps      %ymm2, %ymm1, %ymm1
    addl  %edx, %esi
    addl  %r13d, %r11d
    decl  %r10d
    jne   .LBB0_8
```

Bottom Line: too early for numbers

FINAL THOUGHTS

THE VECTORIZATION FINAL SOLUTION?

- ISPC is by far the best option I have seen to exploit vectorization
  - <u>it gives the programmer an abstract machine model to reason about</u>
  - it warns about inefficient memory accesses (aka gathers and scatters)
- In other words, it's not going to get much easier than that but...
  - full C++ support would be a welcome addition
  - linear algebra operations need to be reimplemented
- ISPC is stable, extremely well documented and open source
- For more information visit http://ispc.github.io

BACKUP

MEMORY MATTERS

- Addressing modes
  - SSEx and AVX1 do not support strided accesses pattern and gather-scatter accesses which force the compiler to generate scalar instructions
  - Even when “fancy” access patterns are supported (e.g. IMCI and AVX2) a penalty is paid
  - <u>Convert your arrays of structures to structures of arrays</u>
- Memory alignment
  - Unaligned memory access may generate scalar instructions for otherwise vectorizable code
  - Vectorized loads from unaligned memory suffer from a penalty
  - The penalty decreased over the years and may become negligible in the future
  - <u>Align data in memory to the width of the SIMD unit if possible</u>