Best Practices: CPU Code Optimization

John E. Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

University of Illinois at Urbana-Champaign

http://www.ks.uiuc.edu/Research/gpu/

Workshop on Accelerated High-Performance Computing in Computational Sciences (SMR 2760),

International Centre for Theoretical Physics (ICTP),

Trieste, Italy, June 3, 2015

Some Easy-to-Follow Code Optimization Pages Worth Visiting

- http://www.compileroptimizations.com/
- http://www.agner.org/optimize/
- http://www.agner.org/random/
- http://openmp.org/mp-documents/OpenMP-4.0-C.pdf
- https://software.intel.com/sites/landingpage/IntrinsicsGuide/
- A Guide to Vectorization with Intel® C++ Compilers

Key Points

- Profile code, identify hot spots…Amdahl's Law
- ***Use the best algorithm:***
  - Code optimization won't save an inefficient algorithm
  - SIMD vectorization won't save an inefficient algorithm
- Optimize **innermost loops** first:
  - Ensure good memory access patterns
  - Replace costly operations with cheaper ones, e.g. replace divides with multiplies by reciprocals
  - Reduce overhead by hoisting redundant operations out of loops
  - Unroll loops to decrease control overhead
  - Use SIMD vector instructions to improve throughput

Example of Amdahl's Law Bottleneck: Time-Averaged Electrostatics, Faster GPU Ends up Bottlenecked by Runtime of Formerly Insignificant CPU Code

NCSA Blue Waters Node Type	**Seconds per trajectory frame for one compute node**
Cray XE6 Compute Node: 32 CPU cores (2xAMD 6200 CPUs)	9.33
Cray XK6 GPU-accelerated Compute Node: 16 CPU cores + **NVIDIA X2090 (Fermi) GPU**	2.25
Speedup for GPU XK6 nodes vs. CPU XE6 nodes	**XK6 nodes are 4.15x faster overall**
Tests on XK7 nodes indicate MSM is CPU-bound with the Kepler K20X GPU. Performance is not much faster (yet) than Fermi X2090 Need to move spatial hashing, prolongation, interpolation onto the GPU, or **greatly speedup CPU implementation…**	**In progress….** **XK7 nodes 4.3x faster overall**

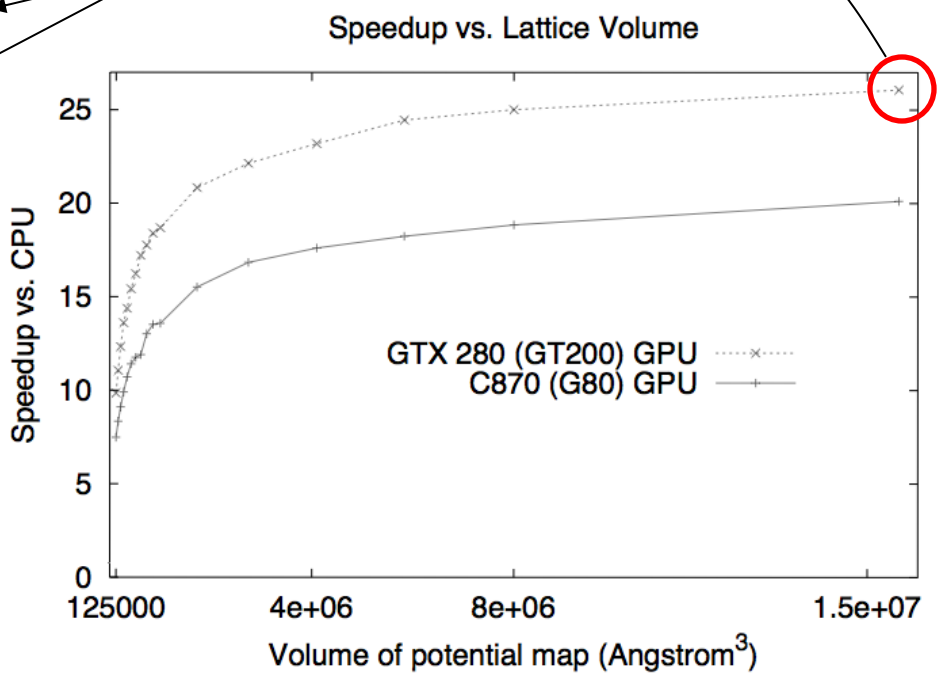
Multilevel Summation on the GPU Ended up Being Limited by CPU, Amdahl's Law Strikes After Several Generations of GPU Speedups

Accelerate **short-range cutoff** and **lattice cutoff** parts

Performance profile for 0.5 Å map of potential for 1.5 M atoms.

Hardware platform is Intel QX6700 CPU and NVIDIA GTX 280.

Computational steps	CPU (s)	w/ GPU (s)	Speedup
Short-range cutoff	480.07	14.87	32.3
Long-range anterpolation	0.18		
restriction	0.16		
lattice cutoff	49.47	1.36	36.4
prolongation	0.17		
interpolation	3.47		
Total	533.52	20.21	26.4



Multilevel summation of electrostatic potentials using graphics processing units.
D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

CPU SIMD Vector Instruction Set Extensions

- ARM NEON
- Intel / AMD x86: MMX, SSE, AVX, AVX-512
- MIPS MDMX
- IBM PowerPC AltiVec
- IBM Power VMX/VSX
- Sun Sparc VIS

Methods for Vectorizing CPU Code

- Compiler autovectorization of **innermost loops** in languages such as C, C++, Fortran, …
  - **Easy to try out!**
  - Works for many simple loop constructs
  - Fails for loops that contain function calls, branching, loop carried dependencies, other issues
- OpenMP 4: #pragma omp simd, …
- Implicit vectorization of sequential code akin to CUDA: OpenCL, Intel SPMD Compiler (ISPC)
- Machine-dependent ***vector intrinsics*** that use a function call syntax to emit machine code

Targets for Vectorization

- Loops are the prime targets for vectorization
  - setup/teardown overheads, e.g. CPU instruction "mode" may have to be changed to vector mode, and back
  - memory alignment requirements
- Rather than vectorizing a single small operation, we typically vectorize a loop of operations
  - Support for multiple vector lengths strongly favors loops rather than single-item scenarios
  - It is ***sometimes*** useful to write functions that operate on a single N-wide vector, but typically only for complex math routines, e.g. rsqrtf(), expf(), and mainly for convenient usage in other loops…

Enable Vector Instructions in Compiler Flags

Intel C/C++:

icpc -m64 –O3 -mavx $(SRC) -o testiccavx

icpc -m64 –O3 -msse2 $(SRC) -o testiccsse

Enable vectorization report:

Old: -vec-report=3

New: -qopt-report=5

g++ -m64 -O3 -mavx $(SRC) -o testgccavx

g++ -m64 -O3 -msse2 $(SRC) -o testgccsse

x86 Vectors: SSE, AVX, AVX-512

- SSE: 128 bit intrinsic data type __m128*
  - 2 double/long, 4 int/float, 8 short, 16 char
- AVX2: 256 bit intrinsic data type __m256*
  - 4 double/long, 8 int/float, 16 short/half, 32 char
- AVX-512: 512 bit intrinsic data type __m512*
  - 8 double/long, 16 int/float, 32 short/half, 64 char

Loops That ***Could*** Autovectorize Well

```
for (i=0; i<N; i++) {
  a[i] = 0.0f;
}

for (i=0; i<N; i++) {
  a[i] = b[i] + c[i] * s;
}
```

Pointer Aliasing and C99 “restrict” Keyword

```
void add(float *a, float *b, float *c) {
  // loops may fail to vectorize because a/b/c may be aliases
  // of each other, and may overlap
}
```

Tell compiler a/b/c are non-overlapping (may need to enable restrict keyword with “-restrict”:

```
void add(float * restrict a, float * restrict b, float * restrict c) {
  // now we have promised the compiler not to give it pointers
  // to a/b/c that overlap, so vectorization is safe…
}
```

Loops That Won't Autovectorize

```
for (i=0; i<N; i++) {
    a[i] = foo(i);    // can't make function calls
}

for (i=0; i<N; i++) {
    if (i==skip)      // can't have branching in loop
        a[i] = 0.0f;
    else
        a[i] = b[i]
}
```

Loops That Won't Autovectorize

```
void vec_dep(int *a, int k, int c, int m) {
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c; // if k is negative, code would break
}
```

```
a[0]=0;
for (i=1; i<N; i++) a[i]=a[j-1]+1; // read-after-write dep
```

```
for (i=1; i<N; i++) a[j-1]=a[j]+1; // write-after-read dep
```

Give Vectorization Hints to the (Intel) Compiler

- Assert that data is [un]aligned
  - #pragma vector [un]aligned
  - __assume_aligned(<*var*>,<*int sz*>)
- Advise compiler of typical loop trip count
  - #pragma loop count ()
- Ignore potential (unproven) data dependences
  - #pragma ivdep
- Override compiler's vectorization efficiency heuristics
  - #pragma vector always

SIMD-Friendly Memory Layout

Conversion From Array-of-Structure to Structure-of-Array

- AOS:

```
typedef struct {
  float x;
  float y;
  float z;
  float w;
} myvec;
myvec aos[1024];
aos[i].x = 0;
aos[i].y = 0;
```

- SOA

```
typedef struct {
  float x[1024];
  float y[1024];
  float z[1024];
  float w[1024];
} myvecs;
myvecs soa;
soa.x[i] = 0;
soa.y[i] = 0;
```

Memory Alignment for Vectorization

- Arrays must be aligned to required boundary for best performance
- Unaligned loads/stores are MUCH slower than aligned loads/stores
- Most OS-provided allocators don't provide required alignment for vectorization
- Local variables (on the stack) are not necessarily aligned, except when special directives or vector types are used

Aligned Memory Allocation

Portable POSIX API:

```
double *a;
posix_memalign((void**) &a, 64, size);
free(a);
```

Intel _mm_malloc() intrinsic:

```
double* a = (double*) _mm_malloc(size, 64);
_mm_free(a);
```

Other Alignment Alternatives

```
/* allocate memory and return a pointer that is aligned on a given   */
/* byte boundary, to be used for page- or sector-aligned I/O buffers */
/* We use this since posix_memalign() is not widely available...     */
/* Definition of integer pointer type for C99 */
#define myintptrtype uintptr_t

void *alloc_aligned_ptr(size_t sz, size_t blocksz, void **unalignedptr) {
  // pad the allocation to an even multiple of the block size
  size_t padsz = (sz + (blocksz - 1)) & (~(blocksz - 1));
  void * ptr = malloc(padsz + blocksz + blocksz);
  *unalignedptr = ptr;
  return (void *) ((((myintptrtype) ptr) + (blocksz-1)) & (~(blocksz-1)));
}
```

Aligning Local Variables

Old: __declspec(align(*n*))

New: __attribute__((align(*n*)))

```
__attribute__((align(64))) double d[] =
                        {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
```

Alignment Macro in VMD (works with various compilers)

```
#if defined(__GNUC__) && ! defined(__INTEL_COMPILER)
#define __align(X)  __attribute__((aligned(X) ))
#else
#define __align(X) __declspec(align(X) )
#endif

 __align(16) mystruct n;
```

Cache Hints, Prefetching

- Use directives or compiler flags to cause compiler to ***prefetch*** data prior to the point at which it will be used, to help the memory system hide latency
- Conversely, it is sometimes helpful to use ***cache bypassing*** loads and stores, when working with streaming data that is too large to fit, or with a very large stride

What If Autovectorization Fails, or Can't be Made Efficient?

- Sometimes vectorization must be done by hand in order to be successful
- To quote an Intel vectorization guide:

"There will be cases when writing your own SIMD code is unavoidable, e.g. when the compiler cannot auto-vectorize or when you can vectorize the code much better than the compiler. "

Performance Evaluation:

Molekel, MacMolPlt, and VMD

Sun Ultra 24: Intel Q6600, NVIDIA GTX 280

	C_{60}-A	C_{60}-B	Thr-A	Thr-B	Kr-A	Kr-B
Atoms	60	60	17	17	1	1
Basis funcs (unique)	**300 (5)**	**900 (15)**	**49 (16)**	**170 (59)**	**19 (19)**	**84 (84)**

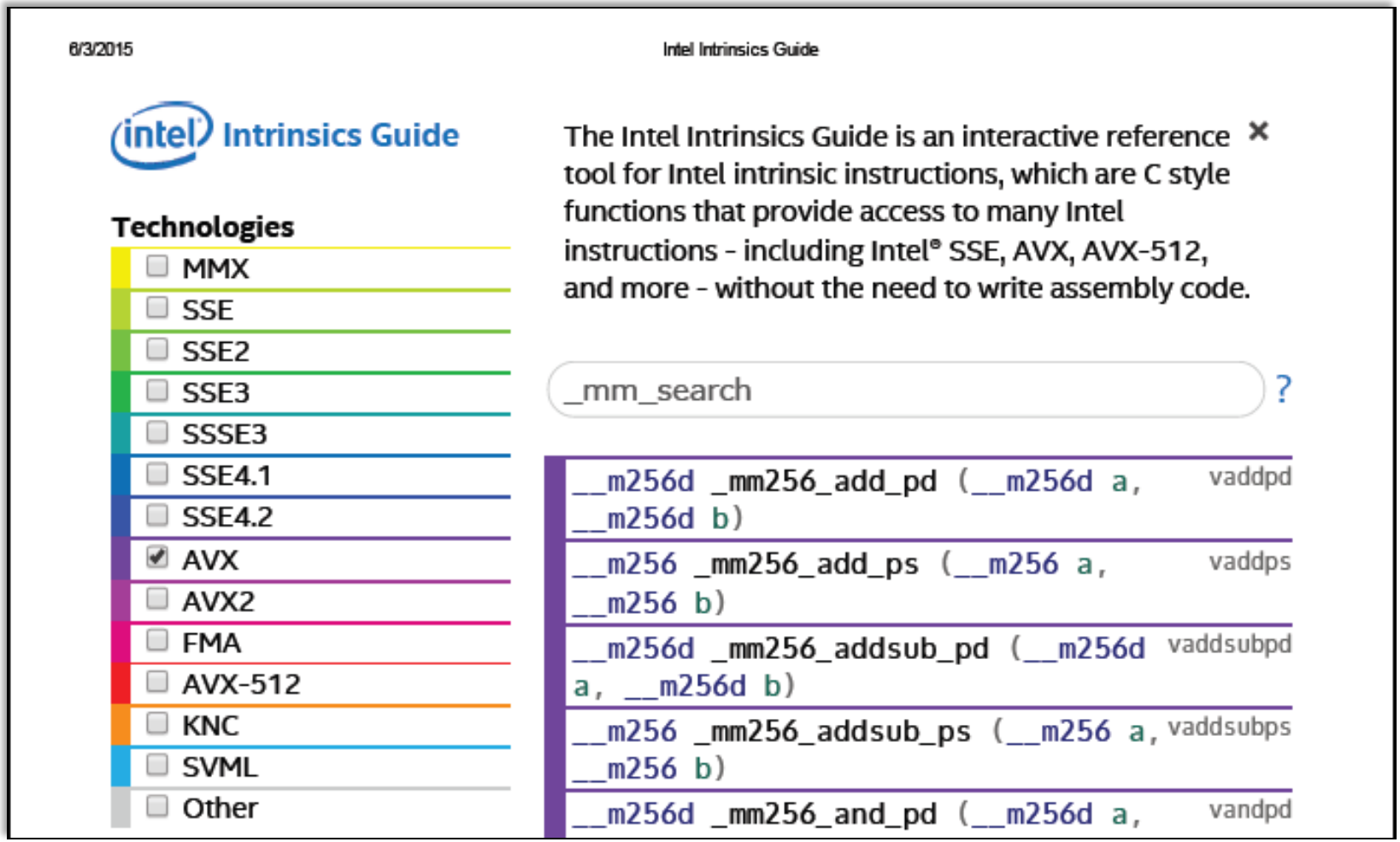
Kernel	Cores GPUs	Speedup vs. Molekel on 1 CPU core					
Molekel	1*	1.0	1.0	1.0	1.0	1.0	1.0
MacMolPlt	4	2.4	2.6	2.1	2.4	4.3	4.5
VMD GCC-cephes	4	3.2	4.0	3.0	3.5	4.3	6.5
VMD ICC-SSE-cephes	4	16.8	17.2	13.9	12.6	17.3	21.5
VMD ICC-SSE-approx**	4	59.3	53.4	50.4	49.2	54.8	69.8
VMD CUDA-const-cache	1	552.3	533.5	355.9	421.3	193.1	571.6

Using Intrinsics for SIMD Instructions

- Set compiler flags to allow generation of SIMD instructions
- #include appropriate headers
- Use memory alignment fctns, declarations
- Make sure memory buffers are padded out, or that loops are safe

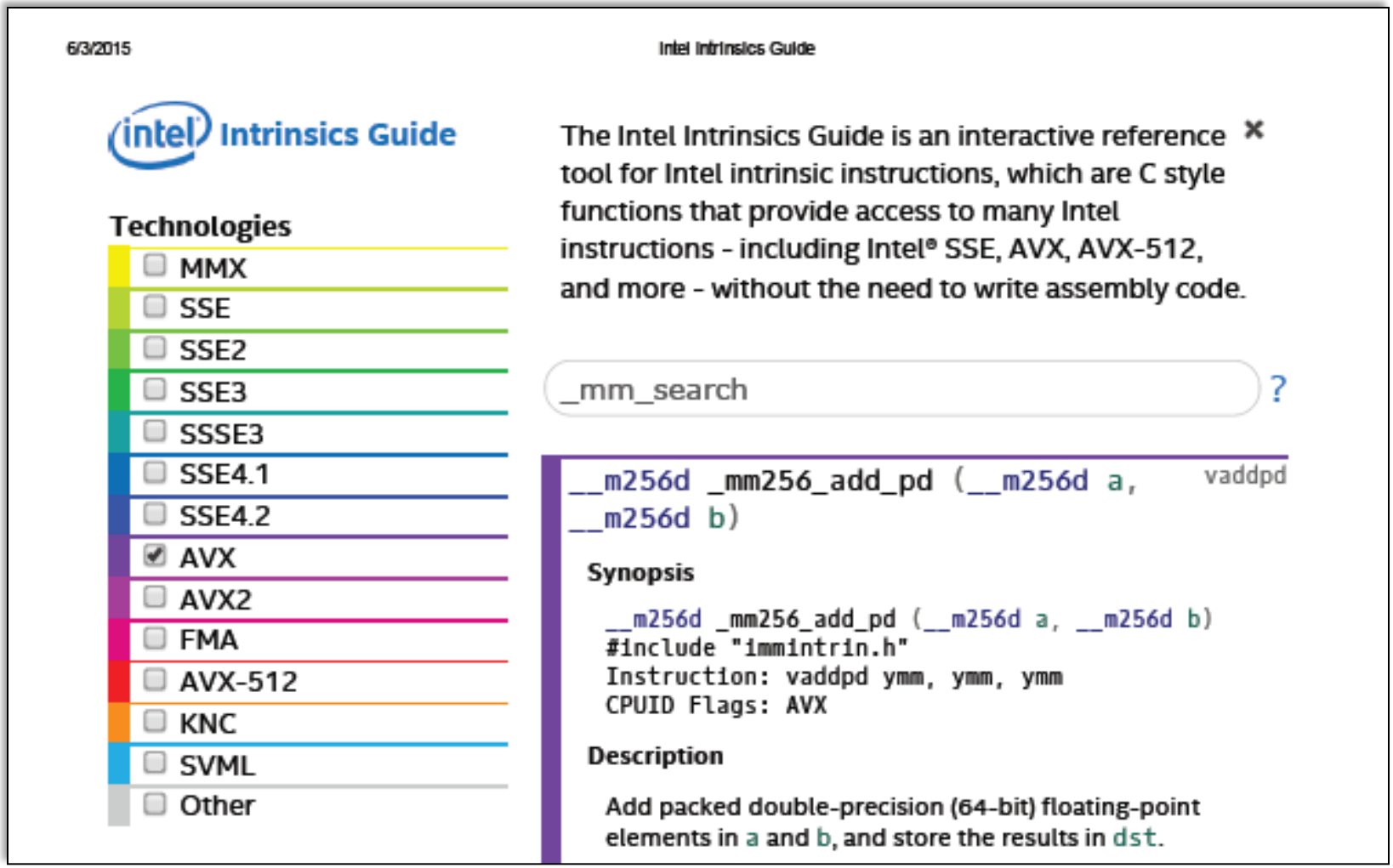
Intel's Interactive Intrinsics Guide

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/



Intel's Interactive Intrinsics Guide

- https://software.intel.com/sites/landingpage/IntrinsicsGuide/



Using Intrinsics Headers

```
#if defined(__SSE2__)
#include <emmintrin.h>
#endif
#if defined(__AVX__)
#include <immintrin.h>
#endif
#if defined(__ARM_NEON__)
#include <arm_neon.h>
#endif
```

Multiply Two Vectors Element-wise

```
float out[] = { 0.0f, 0.0f, 0.0f, 0.0f };
__m128 v4a = _mm_set_ps(1.0f, 2.0f, 3.0f, 4.0f);
__m128 v4b = _mm_set_ps1(2.0f);
__m128 v4out = _mm_mult_ps(v4a, v4b);
_mm_store_ps(out, v4out);
```

Sum of Matrix-Vector Product Plain C

```
// Calculate sum of matrix-vector product
// for 8x8 A and 8x1 u:  v += A u
for (int k=0, j=0;  j < C1_VECTOR_SIZE;  j++) {
    for (int i = 0;  i < C1_VECTOR_SIZE;  i++, k++) {
        v[j] += A[k] * u[i];
    }
}
```

Sum of Matrix-Vector Product AVX

```
__m256 uelem8 = _mm256_load_ps(parms.au);
__m256 v8 = _mm256_set1_ps(0.0f);
for (int j=0;  j < C1_VECTOR_SIZE;  j++) {
    __m256 us8 = _mm256_broadcast_ss(&parms.au[j]);
    __m256 melem8 = _mm256_load_ps(parms.aa + j*8);
    __m256 tmp8 = _mm256_mul_ps(melem8, us8);
    v8 = _mm256_add_ps(v8, tmp8);
}
__m256 newv8 = _mm256_load_ps(parms.av);
newv8 = _mm256_add_ps(newv8, v8);
_mm256_store_ps(parms.av, newv8);
```

Search a List for Non-Zero Values w/ AVX

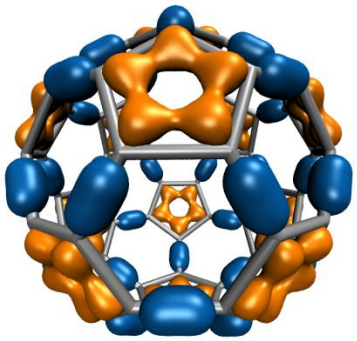
```
int find_first_selection_aligned(int n, const int *on, int *firstsel) {
  int i=0;
[…handling for non-aligned array start goes here…]
  for (; i<(n-7); i+=8) {
    __m256i on8 = _mm256_load_si256((__m256i*) &on[i]); // aligned load of 8 values
    if (!_mm256_testz_si256(on8, on8))
      break; // found a block containing the first selected atom
  }
 for (; i<n; i++) {
    if (on[i]) {
      *firstsel = i; // found first selected atom
      return 0;
    }
  }
 *firstsel = 0; // no atoms were selected if we got here
  return -1;
}
```

SSE Min Example

```
// helper routine to perform a min among all 4 elements of an __m128
static float fmin_m128(__m128 min4) {
  __m128 tmp;
  tmp = min4;
  tmp = _mm_shuffle_ps(tmp, tmp, _MM_SHUFFLE(2, 3, 0, 1));
  tmp = _mm_min_ps(min4, tmp);
  min4 = tmp;
  tmp = _mm_shuffle_ps(tmp, tmp, _MM_SHUFFLE(1, 0, 3, 2));
  tmp = _mm_min_ps(min4, tmp);
  min4 = tmp; // all 4 elements are now set to the min

  float fmin;
  _mm_store_ss(&fmin, min4);
  return fmin;
}
```

Molecular Orbital Inner Loop, Hand-Coded SSE Hard to Read, Isn't It? (And this is the "pretty" version!)

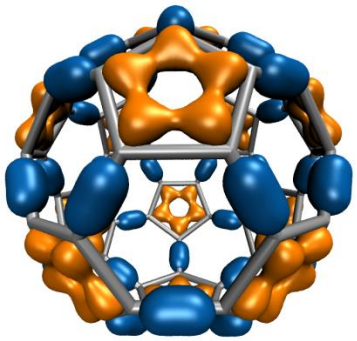


```
for (shell=0; shell < maxshell; shell++) {
  __m128 Cgto = _mm_setzero_ps();
  for (prim=0; prim<num_prim_per_shell[shell_counter]; prim++) {
    float exponent       = -basis_array[prim_counter    ];
    float contract_coeff =  basis_array[prim_counter + 1];
    __m128 expval = _mm_mul_ps(_mm_load_ps1(&exponent), dist2);
    __m128 ctmp = _mm_mul_ps(_mm_load_ps1(&contract_coeff), exp_ps(expval));
    Cgto = _mm_add_ps(contracted_gto, ctmp);
    prim_counter += 2;
  }
  __m128 tshell = _mm_setzero_ps();
  switch (shell_types[shell_counter]) {
    case S_SHELL:
      value = _mm_add_ps(value, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), Cgto));    break;
    case P_SHELL:
      tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), xdist));
      tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), ydist));
      tshell = _mm_add_ps(tshell, _mm_mul_ps(_mm_load_ps1(&wave_f[ifunc++]), zdist));
      value = _mm_add_ps(value, _mm_mul_ps(tshell, Cgto));
      break;
```

Until now, writing SSE kernels for CPUs required assembly language, compiler intrinsics, various libraries, or a really smart autovectorizing compiler and lots of luck...

Molecular Orbital Inner Loop, OpenCL Vec4 Ahhh, much easier to read!!!

```
for (shell=0; shell < maxshell; shell++) {
  float4 contracted_gto = 0.0f;
  for (prim=0; prim < const_num_prim_per_shell[shell_counter];  prim++) {
    float exponent       = const_basis_array[prim_counter    ];
    float contract_coeff = const_basis_array[prim_counter + 1];
    contracted_gto += contract_coeff * native_exp2(-exponent*dist2);
    prim_counter += 2;
  }
  float4 tmpshell=0.0f;
  switch (const_shell_symmetry[shell_counter]) {
    case S_SHELL:
      value += const_wave_f[ifunc++] * contracted_gto;      break;
    case P_SHELL:
      tmpshell += const_wave_f[ifunc++] * xdist;
      tmpshell += const_wave_f[ifunc++] * ydist;
      tmpshell += const_wave_f[ifunc++] * zdist;
      value += tmpshell * contracted_gto;
      break;
```



OpenCL's C-like kernel language is easy to read, even 4-way vectorized kernels can look similar to scalar CPU code. All 4-way vectors shown in green.

Intel ISPC Compiler

- Single-Program Multiple-Data (SPMD) programming language/compiler
- Uses a dialect of C
- ISPC syntax resembles the serial single-thread style of CUDA or OpenCL, but it produces Intel x86 or Xeon Phi vector instructions
- Much easier to write than intrinsics, and overcomes many of the most annoying limitations that plague the traditional autovectorization approaches implemented in Intel C/C++ and GNU GCC
- Compiler is open source and freely available: https://ispc.github.io/ispc.html

Intel ISPC Mandelbrot Example

- https://ispc.github.io/example.html

```
static inline int mandel(float c_re, float c_im, int count) {
    float z_re = c_re, z_im = c_im;
    int i;
    for (i = 0; i < count; ++i) {
        if (z_re * z_re + z_im * z_im > 4.)
            break;
        float new_re = z_re*z_re - z_im*z_im;
        float new_im = 2.f * z_re * z_im;
        z_re = c_re + new_re;
        z_im = c_im + new_im;
    }
    return i;
}
```

Acknowledgements

- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- Funding:
  - NSF OCI 07-25070
  - NSF PRAC “The Computational Microscope”
  - NIH support: 9P41GM104601, 5R01GM098243-02


NIH BTRC for Macromolecular Modeling and Bioinformatics
1990-2017
Beckman Institute
University of Illinois at
Urbana-Champaign