High-Performance X-Ray Scattering Data Analysis

Abhinav Sarje
Computer Research Scientist
Lawrence Berkeley National Laboratory
Berkeley, CA
Background

- **HipGISAXS:** a massively-parallel high-performance grazing incidence small angle X-ray scattering data analysis software.
  - Written in **C++** with **MPI + OpenMP [+ CUDA.]**
  - Double-precision complex number computations.
An Example Kernel

```c
for(int z = 0; z < nqz; ++ z) { // O(10^6)
    int y = z % nqy;
    vector3c_t mq = rotate(qx[y], qy[y], qz[z], rot);
    complex_t qpar = sqrt(mq[0] * mq[0] + mq[1] * mq[1]);
    ... more computations ...

    complex_t temp_ff(0.0, 0.0);
    for(int i_r = 0; i_r < rsize; ++ i_r) { // O(1) - O(10)
        for(int i_h = 0; i_h < hsize; ++ i_h) { // O(1) - O(10)
            ... more computations ...
            complex_t expo_val = exp(0.5 * mq[3] * h[i_h]);
            complex_t sinc_val = sinc(0.5 * mq[3] * h[i_h]);
            complex_t bess_val = cbessj(qpar * r[i_r], 1) / (qpar * r[i_r]);
            temp_ff += sinc_val * expo_val * bess_val;
        }
    }
    ... more computations ...

    complex_t temp2 = exp(temp1);
    ff[z] = temp_ff * temp2;
}
```
Optimizing for Intel Processors: Platforms for Analysis

- **Edison** (Cray XC30) @ NERSC:
  - Intel Ivy Bridge (Xeon E5-2695). 12 cores.

- **Babbage** @ NERSC:
  - Intel Xeon Phi (KNC). 60 cores.
Optimizing for Intel Processors: Threading

- Mostly *embarrassingly-parallel* computations.
- Primary performance analysis tools used:
  - *Intel VTune, TAU, PAPI.*
- Effective threading using **OpenMP:**

![Graph on Edison (Ivy Bridge)](image1)

**On Edison:** (Ivy Bridge)

![Graph on Babbage (MIC/KNC)](image2)

**On Babbage:** (MIC/KNC)
Kernel Vectorization: Attempt 1

- Compiler-based auto-vectorization. (Intel compiler 15.0.)
- Requirements for auto-vectorization:
  - Loop should be single-block, typically without branches/jumps.
  - Loop must be countable.
  - No backward loop-carried dependencies.
  - No special functions or subroutine calls (unless inlined.)
  - Generally inner-most loop in a nest.
Kernel Vectorization: Attempt 1

- **Compiler-based auto-vectorization.** (Intel compiler 15.0.)
- **Requirements for auto-vectorization:**
  - ❌ Loop should be single-block, typically without branches/jumps.
  - ✔ Loop must be countable.
  - ✔ No backward loop-carried dependencies.
  - ✔ No special functions or subroutine calls (unless inlined.)
  - ❌ Generally inner-most loop in a nest.
Kernel Vectorization: Attempt 1

- **Compiler-based auto-vectorization.** (Intel compiler 15.0.)

- **Requirements for auto-vectorization:**
  - ✗ Loop should be single-block, typically without branches/jumps.
  - ✓ Loop must be countable.
  - ✓ No backward loop-carried dependencies.
  - ✓ No special functions or subroutine calls (unless inlined.)
  - ✗ Generally inner-most loop in a nest.

- **Pragmas and explicit directives failed.**
Kernel Vectorization: Attempt 2

- Using Intel Math Kernel Library (MKL):
  - VML and CBLAS (level 1) vector functions.
Kernel Vectorization: Attempt 2

- Using Intel Math Kernel Library (MKL):
  - VML and CBLAS (level 1) vector functions.

- Default VML mode HA (High Accuracy, 1 ulp.)
  - IvyBridge: Time = 2.91x base [speedup = 0.34], # instructions = 3.81x base.
  - MIC: Time = 0.82x base [speedup = 1.23].
Kernel Vectorization: Attempt 2

- Using Intel Math Kernel Library (MKL):
  - VML and CBLAS (level 1) vector functions.

- Default **VML mode HA** (*High Accuracy, 1 ulp.)*
  - *IvyBridge:* Time = 2.91x base [speedup = 0.34], # instructions = 3.81x base.
  - *MIC:* Time = 0.82x base [speedup = 1.23].

- **VML mode LA** (*Low Accuracy, 4 ulp.)*
  - *IvyBridge:* Time = 2.68x base [speedup = 0.37], # instructions = 3.49x base.
  - *MIC:* Time = 0.42x base [speedup = 2.41].
Kernel Vectorization: Attempt 2

- **Using Intel Math Kernel Library (MKL):**
  - VML and CBLAS (level 1) vector functions.

- **Default VML mode HA** (*High Accuracy, 1ulp.*)
  - *IvyBridge:* Time = 2.91x base [speedup = 0.34], # instructions = 3.81x base.
  - *MIC:* Time = 0.82x base [speedup = 1.23].

- **VML mode LA** (*Low Accuracy, 4ulp.*)
  - *IvyBridge:* Time = 2.68x base [speedup = 0.37], # instructions = 3.49x base.
  - *MIC:* Time = 0.42x base [speedup = 2.41].

- **VML mode EP** (*Enhanced Performance, 50% bits accurate.*)
  - *IvyBridge:* Time = 0.50x base [speedup = 1.98], # instructions = 0.61x base.
  - *MIC:* Time = 0.086x base [speedup = 11.64].
Kernel Vectorization: Attempt 3

- Using Intel AVX intrinsics.
  - Implemented complex number operations with intrinsics.
Kernel Vectorization: Attempt 3

- Using Intel AVX intrinsics.
  - Implemented complex number operations with intrinsics.
  - Used **hybrid AoS and SoA**. E.g.:

```c
typedef struct {
    __mm256d real;
    __mm256d imag;
} avx_m256c_t;
```
Kernel Vectorization: Attempt 3

- Using Intel AVX intrinsics.
  - Implemented complex number operations with intrinsics.
  - Used hybrid AoS and SoA. E.g.:

```c
typedef struct {
    __mm256d real;
    __mm256d imag;
} avx_m256c_t;

inline avx_m256c_t avx_mul_ccp(avx_m256c_t a, avx_m256c_t b) {
    avx_m256c_t v;
    avx_m256_t temp1 = _mm256_mul_pd(a.real, b.real);
    avx_m256_t temp2 = _mm256_mul_pd(a.imag, b.imag);
    avx_m256_t temp3 = _mm256_mul_pd(a.real, b.imag);
    avx_m256_t temp4 = _mm256_mul_pd(a.imag, b.real);
    v.real = _mm256_sub_pd(temp1, temp2);
    v.imag = _mm256_add_pd(temp3, temp4);
    return v;
}
```
Kernel Vectorization: Attempt 3

- **Using Intel AVX intrinsics.**
  - Implemented complex number operations with intrinsics.
  - Used **hybrid AoS and SoA.** E.g.:

```c
inline avx_m256c_t avx_mul_ccp(avx_m256c_t a, avx_m256c_t b) {
    avx_m256c_t v;
    avx_m256_t temp1 = _mm256_mul_pd(a.real, b.real);
    avx_m256_t temp2 = _mm256_mul_pd(a.imag, b.imag);
    avx_m256_t temp3 = _mm256_mul_pd(a.real, b.imag);
    avx_m256_t temp4 = _mm256_mul_pd(a.imag, b.real);
    v.real = _mm256_sub_pd(temp1, temp2);
    v.imag = _mm256_add_pd(temp3, temp4);
    return v;
}
```

typedef struct {
    __mm256d real;
    __mm256d imag;
} avx_m256c_t;

- **Performance:**
  - Time = **0.35x** base [speedup = **2.86**], # instructions = **0.28x** base.
Conclusions and Insights

- Auto-vectorization does not always work.
- Intrinsics are best for DP/complex computations.
  - Provide most flexibility in achieving higher performance for non-typical computes.
- Biggest surprises: Intel MKL performance, e.g. v?Exp ( )
  - DP complex, average MKL time = 0.93x base [speedup = 1.08]
  - DP real, average MKL time = 0.53x base [speedup = 1.87]
  - SP complex, average MKL time = 0.34x base [speedup = 2.97]
  - SP real, average MKL time = 0.25x base [speedup = 4.07]
- Would be great if efficient implementations of special functions like Bessel, Sinc were available.
- Already taking Intel’s help.
Acknowledgements

- This work was performed as part of the “Lawrence Berkeley National Lab Intel Parallel Computing Center.”

- This work used resources at NERSC supported by the Office of Science of the U.S. DoE under Contract No. DE-AC02-05CH11231.

- Inputs/suggestions were obtained from Intel folks: Marius Cornea, Jingwei Zhang, CJ Newburn, Hideki Saito.