INTEL SIMD DATA LAYOUT TEMPLATE (INTEL SDLT)

Anoop Madhusoodhanan Prabha

“Thanks to Alex M Wells for this excellent material on Intel SDLT”
Long Story Short..

Object-Oriented code involves modeling collections as Array of Structures (AoS).

AoS data layout leads to lower vectorization efficiency.

Intel SIMD Data Layout Template (Intel SDLT) helps developers stick to their Object-Oriented Design but still get better vectorization efficiency.
Agenda

Performance Problems posed by Object-Oriented Design

Why Intel SDLT?

Components of Intel SDLT

How to use Intel SDLT in your application?

Q&A
Performance Problems posed by Object-Oriented Design

Applications are designed as interaction between objects (Object-Oriented Design).

Every real world entity is modeled as an object (user defined data type like struct or class).

Collection of this above entity will become an Array of Structures (AoS).

```cpp
struct YourStruct
{
    float x;
    float y;
    float z;
};

typedef std::vector<YourStruct> Container;
Container input(count);
Container result(count);
```

Heap Array

- `YourStruct * input = new YourStruct[count];`
- `YourStruct * result = new YourStruct[count];`

Stack Array

- `YourStruct input[count];`
- `YourStruct result[count];`

Vector

- `typedef std::vector<YourStruct> Container;`
- `Container input(count);`
- `Container result(count);`
What’s wrong with AoS? .... SIMD

SIMD allows the same operation to be applied to multiple data elements efficiently with a single instruction.

Unfortunately getting AOS in memory data layout loaded into a vector register is a “strided” load/store operation requiring multiple load/shuffle/insert or gather instructions.

With AOS, as vector register width increases so do the number of instructions required to populate the vector registers.

If the CPU reduces frequency to execute vector instructions (IE: AVX, AVX2) a limited SIMD improvement might not overcome scalar code operating at a higher frequency.
SIMD is effective with Unit Stride Access

If memory layout has multiple instances of a data member adjacent in memory and aligned on a byte boundary matching the vector register width

- Single load/store instruction to move the data into or out of a vector register
- Many SIMD operations can reference an aligned unit-stride memory access as part of the instruction, avoiding a separate load/store instruction altogether

A properly aligned Structure of Arrays (SOA) in memory data layout provides SIMD compatible Unit-Stride memory accesses

SIMD efficiency & speedup can be restored
Issues with SoA integration

Demands for change of the data structure and deviate from Object Oriented Design.

Demands for change of C++ algorithms.

Explicitly handle allocation/freeing of SOA arrays and make sure they are aligned.
What is Intel SDLT?

A C++11 template library providing concepts of Containers, accessors, Offsets, and Indexes to abstract out different aspects of creating an efficient data parallel SIMD program.

- Containers encapsulate the in memory data layout of an Array of “Plain Old Data“ objects.
- SIMD loops use accessors with an array subscript operator (just like C++ arrays) to read from or write to the objects in the Containers.
- Offsets can be embedded in accessors or applied to a Index passed to the accessors array subscript operator.

Since these concepts are abstracted out, multiple concrete versions can exist and can encapsulate best known methods, thus avoiding common pitfalls in generating efficient SIMD code.
Why Intel SDLT?

A C++11 template library providing concepts of Containers, accessors, Offsets, and Indexes to abstract out different aspects of creating an efficient data parallel SIMD program.

SDLT provides a means to preserve the Array of Structure (AoS) interface for the developers but lays out the data in Structure of Array (SoA) format which is more SIMD friendly and increases the vectorization efficiency.

SDLT provides 1D containers which provides the same interface as std::vector making the ease of integration and interoperability easy.

- push_back, resize, erase, insert, size, capacity, swap, etc
- iterator support: begin(), end(), cbegin(), cend(), rbegin(), rend(), etc.

Works well with all STL algorithms

- for_each, find, search, fill, copy, swap, copy_backward, sort, stable_sort, etc
Before and After Intel SDLT enabling

SDLT Containers

What if that std::vector could store data SOA internally while exposing an AOS view to the programmer?

- Primary goal of SDLT Containers is to meet the requirement above.

SDLT Containers abstract the in memory data layout to:

- AOS (Array of Structures)
- SOA (Structure of Arrays)
- ASA (Array of Structures of Arrays)
  - For a given # of Simd Lanes: NS = 2
    - Struct Index: SI = i/NS
    - Lane Index: LI = i%NS
SDLT 1D container

```
typedef sdlt::soa1d_container<YourStruct> Container;
Container inputContainer(count);
Container resultContainer(count);
```

Intent is data be kept in an SOA or ASA Container the entire time instead of converting from AOS.

SDLT's container will internally store the members of YourStruct in a one dimensional "Structure of Arrays" (SOA) layout.

- Places aligned arrays inside a single allocated buffer vs. a separate allocation per array

Just like std::vector the Containers own the array data and its scope controls the life of that data.
SDLT Primitives

How do the Containers discover the data members of your struct?

C++ lacks runtime time reflection, so the user must provide SDLT with some information on the layout of `YourStruct`.

Runtime Reflection enables to understand:

- What is the actual class of the object
- What attributes does the class have
- What are the actual values of these attributes for the objects

This is easily done with the SDLT_PRIMITIVE helper macro that accepts a struct type followed by a list of its data members.

- A struct must be declared as a primitive before it is used as template parameter to a Container.

```
struct Point3s {
  float x;
  float y;
  float z;
};

struct AABB {
  Point3s topLeft;
  Point3s bottomRight;
};

SDLT_PRIMITIVE(Point3s, x, y, z)
SDLT_PRIMITIVE(AABB, topLeft, bottomRight)
```
SDLT Primitive Requirements

Plain Old Data with fixed size with no allocations

All data members need to be a built-in type or another Primitive

No reference data members

No unions

No bit fields

No bool types

- Comparison semantics not efficient in SIMD
- Use 32-bit integer and compare against known values like 0 or 1 explicitly

Data members need to be public or declare SDLT_PRIMITIVE_FRIEND in the object's definition
SDLT Accessor

To separate data ownership semantics from data access, a separate class called an accessor is used to access the transformed data that is owned by the Container.

Just call "access()" or "const_access()" on a container to get back an accessor or const_accessor. All Containers have defined a typedef for their accessor and const_accessor.

```
Container::const_accessor<> input = inputContainer.const_access();
Container::accessor<> result = resultContainer.access();
```

Use the C++11 keyword "auto" to let the compiler deduce the type.

```
auto input = inputContainer.const_access();
auto result = outputContainer.access();
```
SDLT Accessor Contd..

Embedded Offset

```cpp
template
auto input = inputContainer.\texttt{const\_access}();
auto input2 = inputContainer.\texttt{const\_access}(256);
auto input3 = inputContainer.\texttt{const\_access}(sdlt::aligned\_offset\langle8\rangle(256));
auto input4 = inputContainer.\texttt{const\_access}(sdlt::fixed\_offset\langle256\rangle);
```

Subscript operator

```cpp
void setAllValuesTo(
    Container::accessor iValues,
    const YourStruct &iDefaultValue)
{
    for(int i=0; i < iValues.get\_size\_d1(); ++i)
    {
        iValues[i] = iDefaultValue;
    }
}
```
SDLT Accessor Contd..

The subscript operator[index] returns a Proxy Object

The main use of the Proxy Objects is to import/export data to/from a local variable

Can assign local stack instance of YourStruct to the Proxy

```
YourStruct result = ...
iValues[index] = result;
```

Can retrieve YourStruct from the Proxy to a local stack object

```
YourStruct local = iValues[index];
```

SDLT's design makes use of local objects and the compiler's dead code elimination features.

Overloaded +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=, ==, !=, <, >, <=, >=, +, -, *, /, %, !, &&, ||, &, |, ^, !, ~, <<, >> operators.
Combine AoS with Fixed Size SoA

```cpp
int SimdLaneCount = 8;
int structCount = (count + (SimdLaneCount-1))/SimdLaneCount;

template<int SimdLaneCountT, int ByteAlignmentT>
struct SimdYourStruct
{
    float x[SimdLaneCount];
    float y[SimdLaneCount];
    float z[SimdLaneCount];
} __attribute__((aligned(ByteAlignmentT)));

SimdYourStruct<SimdLaneCount, 32> inputASA[structCount];
SimdYourStruct<SimdLaneCount, 32> outputASA[structCount];

for(int structIndex=0; structIndex < structCount; ++structIndex) {
    #pragma simd
    for(int laneIndex=0; laneIndex < SimdLaneCount; ++laneIndex) {
        YourStruct val;
        val.x = yourStructASA[structIndex].x[laneIndex];
        val.y = yourStructASA[structIndex].y[laneIndex];
        val.z = yourStructASA[structIndex].x[laneIndex];
        ...
    }
}
```
sdlt::asa1d_container

```cpp
int SimdLaneCount = 8;

sdlt::asa1d_container<YourStruct, SimdLaneCount> inputASA(count);
sdlt::asa1d_container<YourStruct, SimdLaneCount> outputASA(count);

auto inputs = inputASA.const_access();
auto outputs = outputASA.access();

int structCount = (count + (SimdLaneCount-1))/SimdLaneCount;
for(int structIndex=0; structIndex < structCount; ++structIndex) {
    #pragma simd
    for(int laneIndex=0; laneIndex < SimdLaneCount; ++laneIndex) {
        sdlt::simd_index<> index(structIndex, laneIndex);
        YourStruct val = inputs[index];
        ...
    }
}
```

Requires sdlt::simd_index wrapper to pass structIndex and laneIndex as a single parameter to the subscript operator
Resources

Intel SIMD Data Layout Template Info

- Introducing the Intel SIMD Data Layout Template (Intel SDLT) to boost efficiency in your vectorized C++ code
- Introduction to the Intel SDLT
- Averaging Filter with Intel SDLT
- Boosting the performance of Cartesian to Spherical co-ordinates conversion using Intel SDLT

Code Modernization Links

- Modern Code Developer Community
  - software.intel.com/modern-code
- Intel Code Modernization Enablement Program
  - software.intel.com/code-modernization-enablement
- Intel Parallel Computing Centers
  - software.intel.com/ipcc
- Technical Webinar Series Registration
- Intel Parallel Universe Magazine
  - software.intel.com/intel-parallel-universe-magazine
BACKUP
Possible Overhead When Storing Objects

We are assigning a newBoundary object to a container.

We only want to change the "y" component of the bounds,

- Because we can only import entire objects
  - We must initialize a new Point3ds
  - Transfer the entire object into the container.
    - Will include the "x" and "z" components despite the fact they haven't changed.
  - Because it's an assignment, the compiler can't figure out the values haven't changed.
    - Perhaps another thread had changed the values, and we are reassigning them back.

The point is that it won't eliminate the assignments to the "x" and "z" inside the container.

```cpp
for(int i=0; i < count; ++i) {
  const Point3ds point = points[i];
  const Point3ds boundary = bounds[i];
  if( point.y > boundary.y) {
    Point3ds newBoundary(boundary.x, newpoint.y, boundary.z);
    bounds[i] = newBoundary;
  }
}
```
SDLT Proxy Objects Provide Interface to Data Members

The proxy objects SDLT returns from the [i] operator provide an interface to access the individual data members of the primitive.

The interface provides a method using each data member's name and returns a proxy to that data member for element [i] inside the Container.

```cpp
for(int i=0; i < count; ++i) {
    const Point3ds point = points[i];
    const Point3ds boundary = bounds[i];
    if( point.y > boundary.y ) {
        bounds[i].y() = point.y;
    }
}
```

Now only the "y" component will be updated and the loop is much more efficient.
Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

Copyright © 2016, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

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.

Notice revision #20110804