Intel SIMD Data Layout Template library (Intel SDLT)

Ануфриенко Андрей

"Intel® Software 2017: HPC Code Modernization and Artificial Intelligence"

15 сентября 2017

План

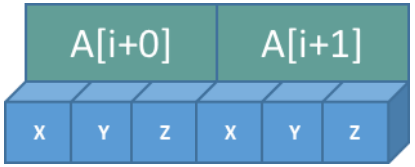
- Проблемы производительности вызываемые объектно-ориентированным дизайном
- Почему Intel® SDLT?
- Компоненты Intel® SDLT
- Как использовать Intel® SDLT в вашем приложении? – Demo
- Q&A

Проблемы производительности вызванные использованием объектно-ориентированного дизайна

- Каждый реальный объект представляет собой определенный пользователем тип данных подобных структуре или классу.

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

- Коллекция таких объектов будет выглядеть как массив структур.
- Например возможные варианты:



- Heap array:

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

- Stack array:

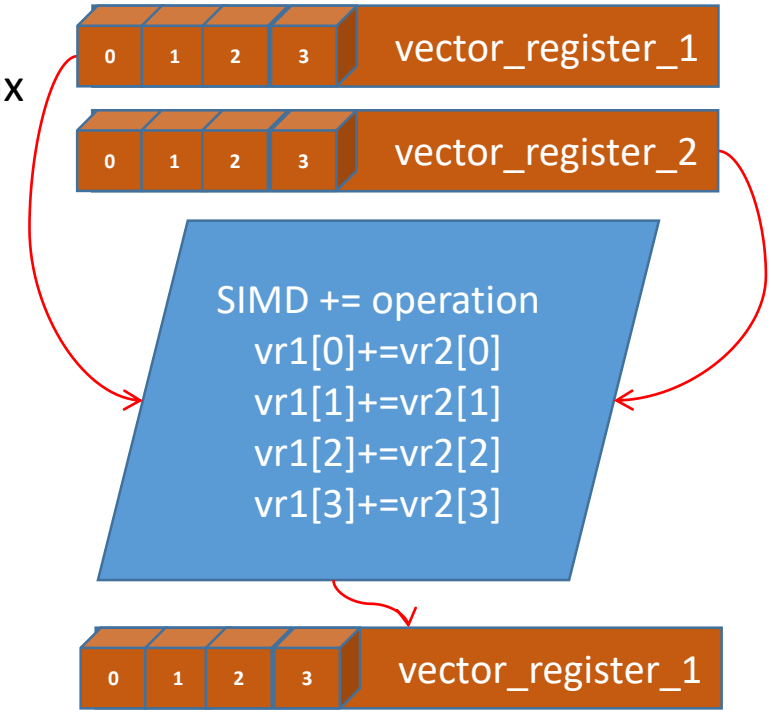
```
    YourStruct input[count];
    YourStruct result[count];
```

- std::vector

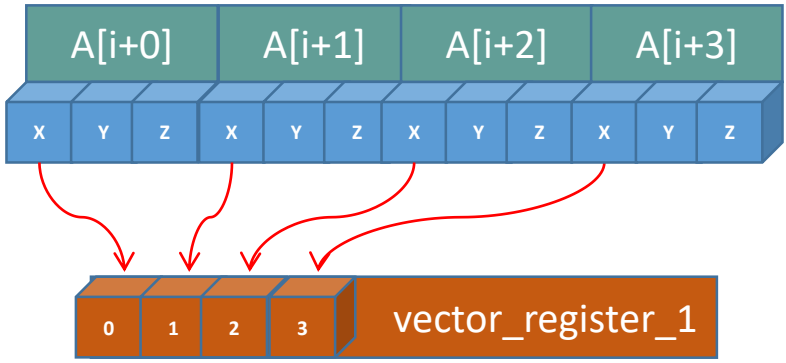
```
typedef std::vector<YourStruct> Container;
Container input(count);
Container result(count);
```

Что не так с AOS?.... SIMD

- SIMD позволяет применять одну и ту-же инструкцию к нескольким элементам данных так же эффективно как одну инструкцию, но для этого данные должны быть упакованы в векторные регистры.
- К сожалению в случае использования непоследовательно хранящихся данных они копируются в векторные регистры поэлементно. Это требует использования нескольких load/shuffle/insert или gather.
- В результате векторизация становится неэффективной, а поскольку в некоторых случаях при выполнении векторных инструкция частота может быть понижена, то такая векторизация может привести и к ухудшению производительности.

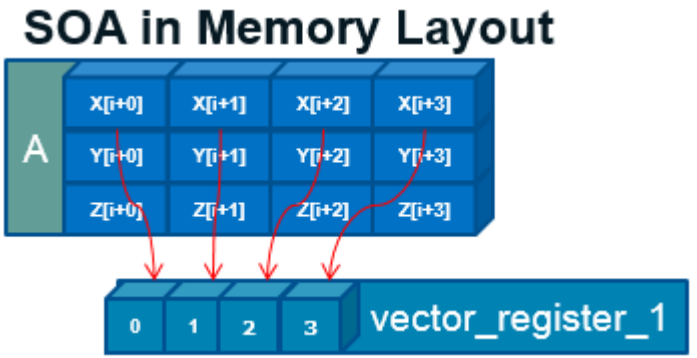


AOS in Memory Layout



SIMD эффективна при последовательном доступе к данным

- Если расположение в памяти нескольких элементов данных непрерывно и выровнено по памяти в соответствии с длиной векторного регистра
  - Может использоваться одна load/store инструкция для перемещения дынных из памяти в регистр и наоборот
  - Многие векторные операции могут иметь такую память в качестве операнда.
- Подходящим образом выравненная в памяти структура массивов (SOA) обеспечивает SIMD совместимую модель доступа к памяти.



Проблемы с реализацией SOA

- Требует пользователя либо отказаться от объектно-ориентированного дизайна объектов либо заводить временные структуры для вычислений.
- Требует изменения C++ алгоритмов.
- Явно управлять аллокацией/освобождением SOA массивов и их выравниванием в памяти.

Что такое Intel® SDLT?

- Это C++11 библиотека шаблонов обеспечивающая реализацию концепции контейнеров, методов доступа, смещений и индексов, призванная абстрагировать различные аспекты создания вычислительных программ эффективно использующих SIMD параллелизм данных.
- SIMD циклы использует аксессоры с индексом массива для чтения и записи объектов в контейнеры.
- Смещения могут быть встроены в аксессорах или применяться к индексу передаваемому оператору аксессора индексу массива.
- Поскольку эти понятия абстрагированы, многочисленные конкретные версии могут существовать и могут инкапсулировать наиболее известные методы, что позволяет избежать типичных ошибок при создании эффективного кода SIMD

Почему Intel® SDLT?

- SDLT обеспечивает средство для сохранения интерфейса AOS (и объектно-ориентированного дизайна) для разработчиков, но выкладывает данные в структуре формата Array (SoA), который является более SIMD дружественным.
- SDLT предоставляет 1D контейнеры, которые обеспечивают такой-же интерфейс как std::vector позволяя легко решать вопросы интеграции и взаимодействия.
  - push_back, resize, erase, insert, size, capacity, swap, etc
  - iterator support: begin(), end(), cbegin(), cend(), rbegin(), rend(), etc.
- Работают хорошо со всеми STL алгоритмами
  - for_each, find, search, fill, copy, swap, copy_backward, sort, stable_sort, etc

SDLT контейнеры

Что было бы если std::vector мог бы хранить данные как SOA, но в то-же время обеспечивая для разработчику привычный объектно-ориентированный дизайн для его приложения?

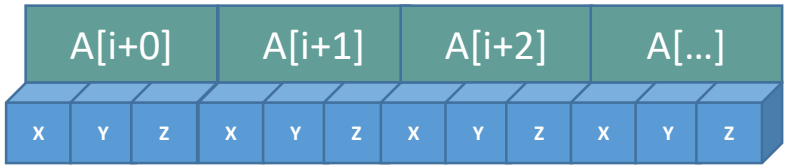
Это и является главной целью SDLT контейнеров.

SDLT контейнеры позволяют хранить данные в памяти в виде:

SOA (Structure of Arrays)

вместо

AOS (Array of Structures)



SDLT 1D container

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

SDLT Примитивы

- Каким образом предоставить контейнеру информацию о ваших структурах?
- Это обеспечивается с помощью SDLT_PRIMITIVE helper macro который принимает структурный тип с последующим списком всех его полей.

```
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 требования для примитивов

- Элементы фиксированного размера без аллокаций
- Все члены данных должны быть встроенные или должны быть другими данными удовлетворяющими этим требованиям.
- Не допускаются ссылочные типы данных
- Не допускаются unions, bit fields, bool types
- Элементы данных должны иметь публичный тип или быть задекларированы как SDLT_PRIMITIVE_FRIEND.

SDLT Accessor

- Accessor используется для доступа трансформированных данных хранящихся в контейнере.
- Используйте C++11 ключевое слово “auto” чтобы позволить компилятору определить тип данных.

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

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

```
void setAllValuesTo(
    Container::accessor iValues,
    const YourStruct &iDefautValue)
{
    for(int i=0; i < iValues.get_size_d1(); ++i)
    {
        iValues[i] = iDefaultValue;
    }
}
```

SDLT интерфейс для доступа к полям структуры (членам класса)

- Интерфейс предоставляет метод использующий имена членов класса для доступа к ним

```
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;
    }
}
```

Пример

```
#include <stdio.h>
#define N 1024
typedef struct RGBs {
   float r;    float g;    float b;
} RGBTy;
void main() {
   RGBTy a[N];
   #pragma omp simd
   for (int k = 0; k<N; ++k) {
      a[k].r = k*1.5;  // non-unit stride access
      a[k].g = k*2.5;  // non-unit stride access
      a[k].b = k*3.5;  // non-unit stride access
   }
   std::cout << "k =" << 10 << ", a[k].r =" << a[10].r <<        ", a[k].g =" << a[10].g <<
     ", a[k].b =" << a[10].b << std::endl;
}
```

Пример

```
#include <stdio.h>
#include <sdlt/sdlt.h>
#define N 1024
typedef struct RGBs {    float r;     float g;     float b; } RGBTy;
SDLT_PRIMITIVE(RGBTy, r, g, b)
void main() {
   sdlt::soa1d_container<RGBTy> aContainer(N);
   auto a = aContainer.access();
#pragma omp simd
   for (int k = 0; k<N; k++) {
      a[k].r() = k*1.5;
      a[k].g() = k*2.5;
      a[k].b() = k*3.5;
   }
   std::cout << "k =" << 10 <<       ", a[k].r =" << a[10].r() <<       ", a[k].g =" << a[10].g() <<
      ", a[k].b =" << a[10].b() << std::endl;
}
```

Additional Resources

- 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(R) SDLT

QA