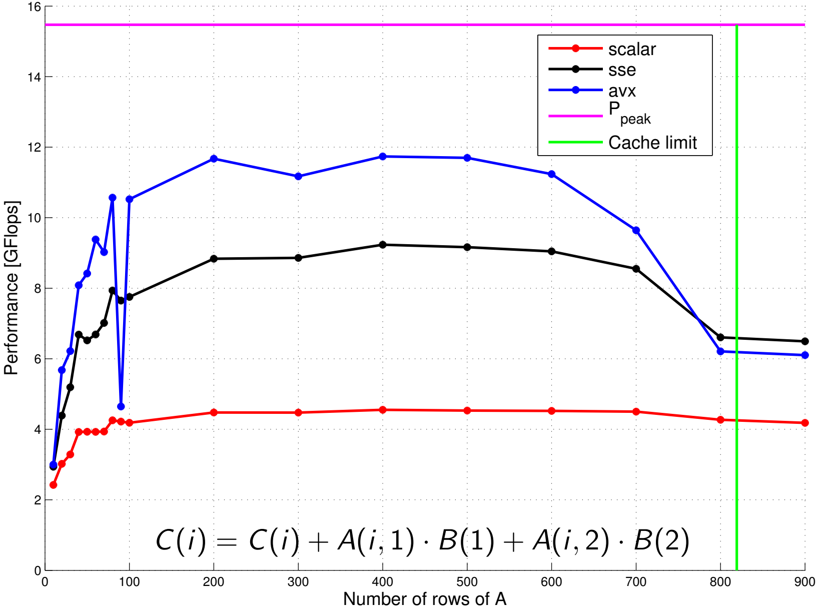
Auto-vectorization

Adam Kosiorek

Supervised by: Prof. M. Gerndt

June 3, 2015

Why do we need vectorization?



$$C(i) = C(i) + A(i,1) \cdot B(1) + A(i,2) \cdot B(2)$$

What is vectorization?

$$\textit{Vectorization} = \textit{Loop unrolling} + \textit{packed SIMD instructions} \tag{1}$$

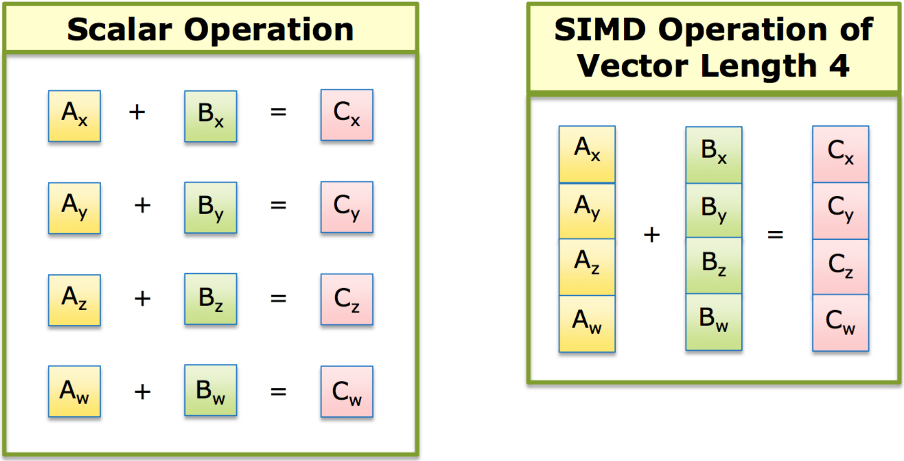
Loop unrolling: manually or by compiler

SIMD: SSE, AVX, Nano etc.

How to get them:

- write assembly
- compiler intrinsics
- special purpose language extensions eg. OpenCL, CUDA
- vectorizing compiler + guidelines

What is SIMD?



Scalar Operation

$A_x + B_x = C_x$

$A_y + B_y = C_y$

$A_z + B_z = C_z$

$A_w + B_w = C_w$

SIMD Operation of Vector Length 4

$$\begin{bmatrix} A_x \\ A_y \\ A_z \\ A_w \end{bmatrix} + \begin{bmatrix} B_x \\ B_y \\ B_z \\ B_w \end{bmatrix} = \begin{bmatrix} C_x \\ C_y \\ C_z \\ C_w \end{bmatrix}$$

The simplest case

```
double A[1024], B[1024], C[1024];
// initialize A and B
for(int i = 0; i < 1024; ++i)
  C[i] = A[i] * B[i];
```

vectorized:

- -O2 and beyond
- speedup $\in [2, 8]$

not vectorized:

- -O0, -O1, -Og, -g, -no-vec
- operates on single entry at a time
- slow

The simplest case - vectorization

default:

```
for(i = 0; i < MAX; ++i)
  C[i] = A[i] * B[i];
```

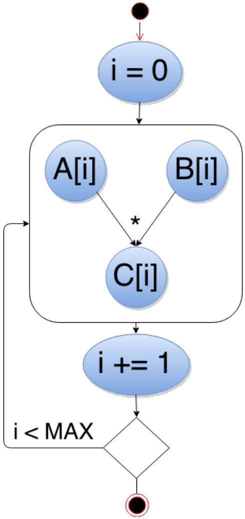
unrolled:

```
for(i = 0; i < MAX; i+=4) {
  C[i] = A[i] * B[i];
  C[i+1] = A[i+1] * B[i+2];
  C[i+2] = A[i+1] * B[i+2];
  C[i+3] = A[i+3] * B[i+3];
}
```

vectorized: (with AVX intrinsics)

```
__m256* mA = (__m256*)A;
__m256* mB = (__m256*)B;
__m256* mC = (__m256*)C;
for(i = 0; i < MAX/4; ++i)
  mC[i] = _mm256_add_pd(mA[i], mB[i]);
```

Why was it vectorized?



Properties needed for vectorization:

- no data dependencies
- single entry, single exit
- no branching
- the innermost loop
- no function calls

Data Dependencies

Read-after-write (flow dependency): ×

```
for(i=1; i<MAX; ++i)
  A[i] = A[i-1] + 1;
```

Write-after-read (anti-dependency): ✓

```
for(i=1; i<MAX; ++i)
  A[i-1] = A[i] + 1;
```

Write-after-write (output dependency): ×

```
for(i=0; i<MAX; ++i)
  A[i - i%2] = A[i] * B[i];
```

Reduction: ✓

```
double sum = 0;
for(i=0; i<MAX; ++i)
  sum = sum + A[i];
```

Assumed Data Dependencies - Pointer Aliasing

```
void compute(int* A, int* B, int* C, int N) {
  for(int i = 1; i<N; ++i)
    C[i] = A[i-1] + B[i];
}
```

- C/C++ makes no assumptions about pointers:

  ```
  compute(a, b, a);
  ```

  is legal!

Single entry, single exit

(b) multiple exits:

```
for(i = 0; i < MAX; ++i)
{
  C[i] = A[i] * B[i];
  if(A[i] < B[i])
    break;
  else
    MAX -= 1
 }
```

(a) single exit:

```
for(i = 0; i < MAX; ++i)
  C[i] = A[i] * B[i];
```

- “Skip” the termination condition
- Modify the termination condition

No branching

(a) no branching?:

```
for(i = 0; i < MAX; ++i)
  if(A[i] != 0)
    C[i] = A[i];
  else
    C[i] = B[i];
```

(b) branching:

```
for(i = 0; i < MAX; ++i)
  switch(i % 3) {
  case 0: C[i] = A[i];
          break;
  case 1: C[i] = B[i];
          break;
  default: C[i] = 0;
  }
```

If statements implemented by “masking assignment” are ok.

Function Calls

```
int compute(int a, int b);
...
for(i = 0; i < MAX; ++i)
  C[i] = compute(A[i], B[i]);
```

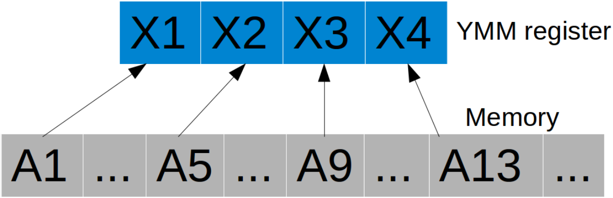
works if the function:

- can be inlined
- is declared as a vector function:

  ```
  __attribute__((vector))
  int compute(int a, int b);
  ```

- is compiler intrinsic function
- is one of the math functions: sin, cos, exp, pow, etc.

Non-contiguous Memory Access



- each entry loaded separately
- happens with non-unit stride and indirect adressing e.g.

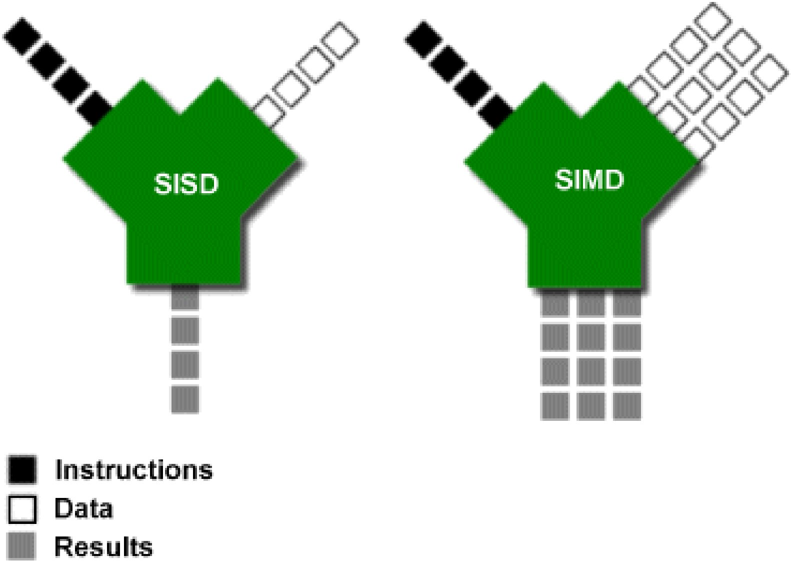
```
C[i] = C[i] * A[i * 2];
C[i] = C[i] * A[B[i]];
```

Note: supported by Intel AVX2

References

1. A Guide to Vectorization with Intel C++ Compilers
   https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers
2. Intrinsics Guide
   https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Questions?


SISD
SIMD
Instructions
Data
Results