CGT 581I - Parallel Graphics and Simulation

Knights Landing Vectorization

Bedrich Benes, Ph.D.
Professor
Department of Computer Graphics
Purdue University

Advanced Vector eXtensions

- Vectorization:
  Execution of 16 single or 8 double precision math operations **at once**

- AVX-512
- Set of vectorization operations
- Can be used
  - automatically (compiler)
  - or enforced (programmer – pragmas)

Optimization report

- Two helpers
  - Vector Advisor analyzer
  - Optimization report

- Intel Compiler has an option

-qopt-report=n

that will generate *.optrpt file

Optimization report

- Long and detailed

```
Report from: Code generation optimizations
Hardware registers
Reserved     :    2[ rsp rip]
Available    :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
Callee-save  :    6[ rbx rbp r12-r15]
Assigned     :    8[ rax rdx rcx rsi rdi r8 r12-r13]
Routine temporaries        Total        :      67
Global    :      16
Local     :      51
Regenerable   :      21
Spilled       :       2
Routine stack
Variables     :     286 bytes*
Reads     :       4 [3.00e+00 ~ 3.4%]
Writes    :       8 [6.00e+00 ~ 6.9%]
Spills        :       0 bytes*
Reads     :       0 [0.00e+00 ~ 0.0%]
Writes    :       0 [0.00e+00 ~ 0.0%]
```

Optimization report

- Parts to pay attention to:

```
Begin optimization report for: Sum(float *, float *, float *, long)

    Report from: Interprocedural optimizations [ipo]

INLINE REPORT: (Sum(float *, float *, float *, long)) [2] main.cpp(15,52)

    Report from: OpenMP optimizations [openmp]

main.cpp(18:1-18:1):OMP:_Z3SumPfS_S_l:  OpenMP DEFINED REGION WAS PARALLELIZED
main.cpp(26:1-26:1):OMP:_Z3SumPfS_S_l:  OpenMP DEFINED REGION WAS PARALLELIZED

   Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]


LOOP BEGIN at main.cpp(29,3)
   remark #15523: loop was not vectorized: loop control variable i was found, but loop
iteration count cannot be computed before executing the loop LOOP END

    Report from: Code generation optimizations [cg]

main.cpp(15,52):remark #34051: REGISTER ALLOCATION : [_Z3SumPfS_S_l] main.cpp:15
```


How to vectorize?

Design your code with vectorization in mind by using:

1) Libraries

2) Auto vectorization

3) SIMD directives

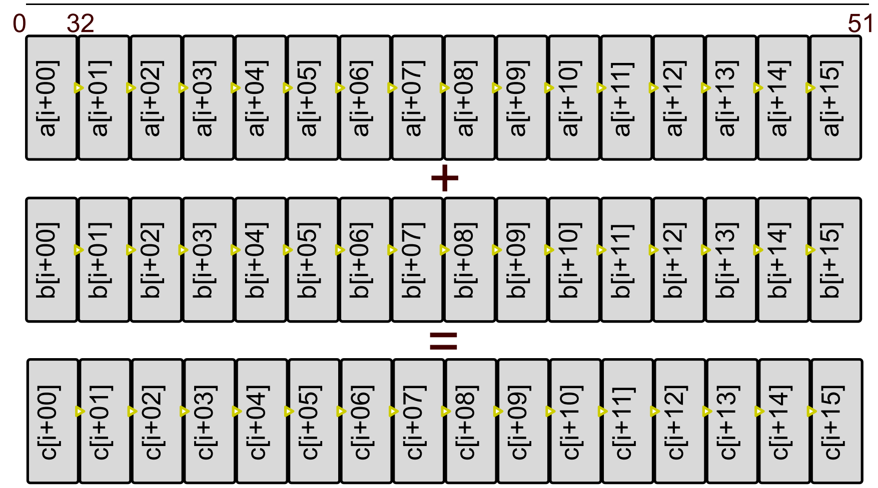
SIMD directives example

- Simple vectorization

```
_declspec(align(16)) float a[max], b[max], c[max];
for (i=0;i<MAX;i++)
 c[i]=a[i]+b[i];
```

`__declspec` is a C++ 11 construct

SIMD directives example



SIMD Example

```
__declspec(align(64)) float X[1000], X2[1000];

void foo(float * restrict a, int n, int n1, int n2) {
  int i;
  __assume_aligned(a, 64);
  __assume(n1%16==0);
  __assume(n2%16==0);

  for(i=0;i<n;i++) { // Compiler vectorizes loop with all aligned accesses
    X[i] += a[i] + a[i+n1] + a[i-n1]+ a[i+n2] + a[i-n2];
  }

  for(i=0;i<n;i++) { // Compiler vectorizes loop with all aligned accesses
    X2[i] += X[i]*a[i];
  }
}
```


© https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

Data Layout

- Vector parallelism
  the same operation on multiple pairs of data
- AVX-512 uses ZMM registers
  https://software.intel.com/en-us/blogs/2013/avx-512-instructions
- The data needs to flow in and out for maximum performance

Gather vs. Scatter

- *Gather*
  applications that are typically grid centric and accumulate values in the vertices (Eulerian approach to CFD)
- *Scatter*
  tasks are location oriented (particles) and measure contribution of the particle to the neighborhood (Lagrangian approach)

Data Layout

- Data aligned and packed in memory
  - 16 floats can be loaded by a single instruction
  - unaligned data require more instructions
  - gather - scatter and permute strategies
  - extra instructions can have performance hit

Data Layout

- Data locality
  - Fetch from cache not memory if possible
    - Memory prefetch to L2, L2 to L1
    - Hardware or software prefetch
    - Inserted by compiler automatically
or
use `mm_prefetch` intrinsics
  - Data reuse
    - Temporal locality – data used later can be prefetched
    - Streaming stores – disable caching for write only

Data Alignment

- Aligned dynamic memory allocation

```
void *_mm_malloc(int size, int base)
```

for MCDRAM:

```
int hbw_posix_memalign(void **memptr,
                       size_t alignment,
                       size_t size)

int hbw_posix_memalign_psize(void **memptr,
                       size_t alignment,
                       size_t size,
                       int pagesize)
```


Data Alignment

- Aligned static memory allocation

```
__attribute__((align(64))) float a[MAX];     //linux gcc

__declspec(align(64)) float a[MAX];          //windows
```

the Intel compiler accepts both

there is also a compiler directive (align)

Data Alignment

- Two step process
  - Align the data
  - Let the compiler know

- Letting know:

```
__assume_aligned(variable, bits)
```

Data Alignment

- Example:

```
void foo(double x[], double y[],int n){
 __assume_aligned(x,64);
 __assume_aligned(y,64);
 for (int i=0;i<n;i++;)
  x[i]=y[i]/2.0;
}
```


Prefetching - compiler

- Hardware prefetchers are efficient
- Compiler can be set to

```
-opt-prefetch=n        //linux
/Qopt-prefetch:n       //Win
```

n=1,2,..,4

four being the most aggressive

Prefetching - pragma

- Compiler prefetch
- Compiler can be set to

```
#pragma prefetch var:hint:distance
```

hint 0 for L1 cache
1 for L2

Prefetching – pragma - example

```
#pragma prefetch tab:1:30
#pragma prefetch tab:0:6
//vprefetch1 for tab with a distance of 30 vectorized iterations ahead
//vprefetch0 for tab with a distance of 6 vectorized iterations ahead
//If pragmas are not present, compiler chooses both distance values

for (i=0; i<2*n; i++) {
 tab[i*max1 + k] = i;
}
```

Prefetch - pragma, noprefetch

- variables can be explicitly not prefetched

```
#pragma noprefetch b
#pragma prefetch a
for(i=0; i<MAX; i++) {
 a[i]=b[i]+1;
}
```


Prefetch - manual

- manual prefetching
- use only after all others failed..
- disable compiler prefetch
- can lead excessive memory communication

```
void _mm_prefetch(char const *address, int hint)
hint: _MM_HINT_T1, _MM_HINT_T0
```


Streaming Stores

- instructions for filling a continuous stream without gaps
- does not use caches

```
Linux and Mac OS X:    -opt-streaming-stores keyword
Windows:               /Qopt-streaming-stores:keyword
keyword
 always  the compiler optimizes
 never
 auto   lets the compiler decide which instructions to use (default)
```


#pragma vector nontemporal

- generates nontemporal hints on stores

```
float a[1024];
void foo(int N){
  int i;
  #pragma vector nontemporal
  for (i = 0; i < N; i++) {
    a[i] = 1;
  }
}
```

#pragma vector nontemporal

- generates nontemporal hints on stores

```
double A[1000];
double B[1000];
void foo(int n){
  int i;
#pragma vector nontemporal (A, B)
  for (i=0; i<n; i++){
    A[i] = 0;
    B[i] = i;
  }
}
```


#pragma vector aligned

- informs compiler the data is aligned

```
void vec_aligned(float *a, int m, int c) {
  int i;
  // Instruct compiler to ignore assumed vector dependencies.
  #pragma vector aligned
  for (i = 0; i < m; i++)
    a[i] = a[i] * c;
  // Alignment unknown but compiler can still align.
  for (i = 0; i < 100; i++)
    a[i] = a[i] + 1.0f;
}
```


#pragma vector always

```
void vec_always(int *a, int *b, int m) {
  #pragma vector always
  for(int i = 0; i <= m; i++)
    a[32*i] = b[99*i];
}
```


When will a loop vectorize?

- if nested, must be the inner one
  (outer loops are auto parallelized OpenMP)
- Must contain a single strain-code line
  (no jumps or branches)
- The loop must be countable
  (the # is known before the loop starts)
- no backward loop-carried dependencies

When will a loop vectorize?

```
for (i=0;i<MAX;i++){
 a[i]=b[i]+c[i];
 d[i]=e[i]-a[i-1]; //OK a[i-1] is valid
}
```

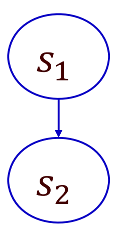
```
for (i=0;i<MAX;i++){
 d[i]=e[i]-a[i-1];
 a[i]=b[i]-c[i];//not OK a[i-1] is needed
}
```


Data Dependence and Loops

- Given two program statements s_1 and s_2
- We say s_2 depends on s_1 if they share a memory location and one of them writes there

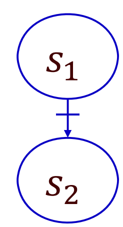
Flow (true) dependence

- Ex:
  s_1 : x=a+b;
  s_2: c=x*10;



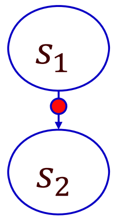
Anti dependence (backward)

- Cannot be reordered
- Ex:
  s_1 : a=x+b;
  s_2: x=c+d;



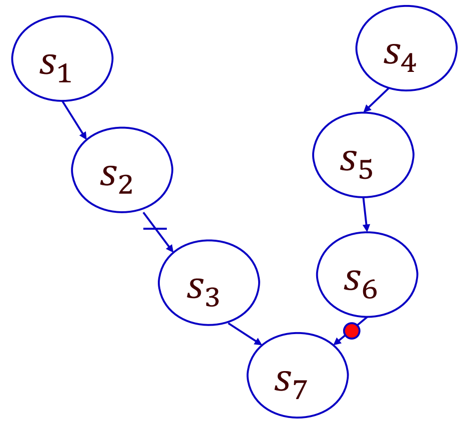
Output dependence

- Ex:
  s_1 : x=a+b;
  s_2: x=c+d;



Parallel vs Sequential

s_1, s_2, s_3 can be executed in parallel with s_4, s_5, s_6:



Loop Data Dependencies

- Example of loop data dependency

```
for (i=0;i<n;i++){
  a[i]=b[i]/2.f;
  c[i]=c[i+1]*a[i];
}
```


How to fix loops?

- Loop Interchange
- Loop Fission
- Loop Fusion
- Loop Peeling
- Loop Unrolling
- Loop Reversal

Loop Interchange

- perfectly nested loops
- no mutual dependency

```
for (int i=0;i<n;i++)
for (int j=0;j<n;j++)
   sum+=a[i][j];
```

```
for (int j=0;j<n;j++)
for (int i=0;i<n;i++)
   sum+=a[i][j];
```


Loop Fission (Loop Splitting)

```
for (int i=0;i<n;i++){
   b[i]=i;
   for (int j=0;j<n;j++)
{
    a[i][j]+=a[i-1][j];
    sum+=a[i][j];
   }
}
```

```
for (int i=0;i<n;i++)
   b[i]=i;
for (int i=0;i<n;i++)
for (int j=0;j<n;j++){
    a[i][j]+=a[i-
   1][j];
     sum+=a[i][j];
}
```


Loop Fission (Loop Splitting)

- possible for loops with independent parts
- changes the order of the execution and can help alignment, caches, etc.

Loop Fusion

- The inverse of loop fission
- Reduces granularity
- Allows more work to be done in parallel on one thread

Loop Peeling

- Takes the first(s) and/or the last(s) iterations
- Enforces initializations

```
for (int i=0;i<n;i++){
    c[i]=a[i]+b[i];
}
```

```
c[0]=a[0]+b[0];
for (int i=1;i<n-1;i++){
    c[i]=a[i]+b[i];
}
c[n-1]=a[n-1]+b[n-1];
```


Loop unrolling

- Carefully!
  The compiler is efficient at vectorization if a loop is **not** manually unrolled
- Unrolling:

```
void Loop(int *a, int *b) {
 for(int i = 0; i <= 3; i++)
    a[i] = b[i];
}
```

```
void Unrolled(int *a, int *b) {
 a[0] = b[0];
 a[1] = b[1];
 a[2] = b[2];
}
```


Loop unrolling

- Let the compiler do it
- n – how many times

```
#pragma unroll n
#pragma nounroll
```


Loop unrolling

```
void unroll(int *a, int *b,
            int *c, int *d) {
  #pragma unroll(4) //inner loop
  for (int i = 1; i < 100; i++) {
    b[i] = a[i] + 1;
    d[i] = c[i] + 1;
  }
}
```

© https://software.intel.com/en-us/node/524556

Loop unrolling

```
int m = 0;
int dir[4]= {1,2,3,4};
int data[10];
#pragma unroll (4)  // outer loop unrolling
 for (int i = 0; i < 4; i++) {
   for (int j = dir[i]; data[j]==N; j+=dir[i])
        m++;
  }
```

© https://software.intel.com/en-us/node/524556

Loop Reversal

- Runs the loop backwards
- Valid only for independent variables

Memory Disambiguation

- Assume

```
void foo(float *a, float *b, float *c, int n){
 for (int i = 0;i<n;i++){
   a[i]=c[i]*b[i];
   b[i]=a[i]+c[i];
}
```

- The compiler does NOT assume a, b, c are independent! No vectorization at all!!
- `a[2]` could be `c[0]`

Memory Disambiguation

- You can let the compiler know

```
void foo(float *restrict a,
         float *restrict b,
         float *restrict c, int n){
 for (int i = 0;i<n;i++){
   a[i]=c[i]*b[i];
   b[i]=a[i]+c[i];
}
```

Memory Disambiguation

- You can let the compiler know that they are not dependent by **ivdep**

```
void foo(float *a, float *b, float *c, int n){
#pragma ivdep
 for (int i = 0;i<n;i++){
   a[i]=c[i]*b[i];
   b[i]=a[i]+c[i];
}
```


#pragma ivdep

- Example:
- The code will NOT vectorize automatically
- *k* is not known and indexing invalid for k<0

```
void indep(int *a, int k, int c, int n){
 for (int i = 0;i<n;i++)
    a[i]=a[i+k]+c;
}
```


#pragma ivdep

- If YOU know, you can tell

```
void indep(int *a, int k, int c, int n){
 #pragma ivdep
 for (int i = 0;i<n;i++)
    a[i]=a[i+k]+c;
}
```


#pragma ivdep

- Another example when ivdep will help

```
#pragma ivdep
 for (i = 0;i<n;i++)
    a[b[i]]=a[b[i]]+1;
```

#pragma omp simd

- Enforces loop vectorization
- Not restricted to inner loops

```
#pragma omp simd
 for (int i = 0;i<n;i++){
   while (a[i]>threshold)
     a[i]*=0.5;
}
```


#pragma omp simd

- Can be combined with parallel for

```
char foo(char *a, int n){
int k;
char c=0;
#pragma omp parallel for simd
 for (k=0;k<n;k++){
   c=c+a[k];
 return c;
}
```


vector always, ivdep, and simd

- **#pragma ivdep**
  - checks for some dependencies
- **#pragma vector always**
  - will not vectorize if there is no gain
- **#pragma simd**
  - does not check for any dependencies you ae responsible
  - will always vectorize

When will SIMD vectorize?

- Countable loops (no break, continue inside)
- Weird non-math operators
- Complex array subscripts
- Low number of iteration loops
- Large loop bodies
- Some C++ exception handlings

fast vs precise math

- `-fp-model=fast`
  - is a compiler directive for fast math
  - this will very likely vectorize
- `-fp-model=precise`
  - is a compiler directive for precise math
  - this may not vectorize
  - parallel computation may not guarantee the same reusuts

SIMD directive clauses

- SIMD has additional clauses
- **PRIVATE, LASTPRIVATE, LINEAR, REDUCTION**
- See Chapter 9 for details

SIMD vectorization

- Be careful…
- SIMD will always vectorize,
  but it does not mean it will be good
- You may not achieve **automatic** vectorization by other pragmas, SIMD will force it

Compiler summary

Syntax of Hint	Semantics
#pragma ivdep	discard assumed data dependences
#pragma vector always	override efficiency heuristics
#pragma vector nontemporal	enable streaming stores
#pragma vector [un]aligned	assert [un]aligned property
#pragma novector	disable vectorization
#pragma distribute point	suggest point for loop distribution
#pragma loop count (<int>)	estimate trip count
restrict	assert exclusive access through pointer
_declspec(align(<int>,<int>))	suggest memory alignment
__assume_aligned(<var>,<int>)	assert alignment property

from
https://software.intel.com/en-us/articles/vectorization-with-the-intel-compilers-part-i

Reading

- Intel Xeon Phi Processor High Performance Programming Knights Landing Edition
- James Jeffers, James Reinders, and Avinash Sodani
- ISBN: 9780128091944
- Morgan Kaufmann
- Chapter 9