Vector Optimizations

Sam Williams
CS 265
samw@cs.berkeley.edu

Topics

- Introduction to vector architectures
- Overview of data dependence analysis
- Loop transformations
- Other optimizations

Introduction

- Goal is to exploit the processing capabilities of vector machines.
- e.g. Transform

```
for(x=0;x<N;x++){
  a[x]=b[x]+c[x]
}
```

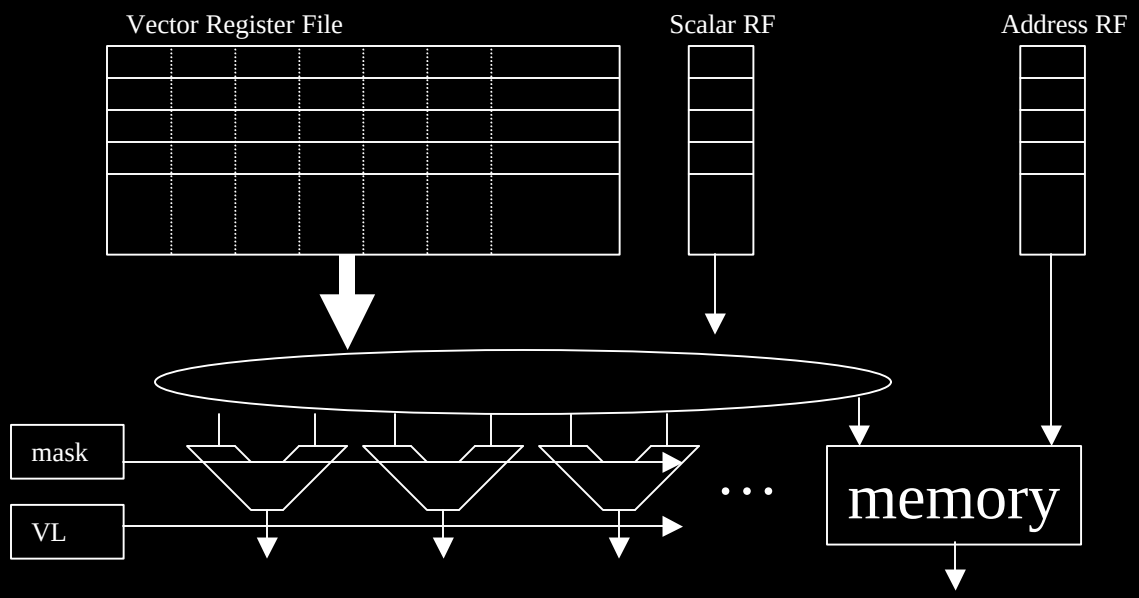
```
    li t3,0
    ld t0,[a]
    ld t1,[b]
    ld t2,[c]
start:
    ld a1,t1
    ld a2,t2
    add a0,a1,a2
    sd a0,t0
    addi t0,t0,8
    addi t1,t1,8
    addi t2,t2,8
    addi t3,t3,1
    bne t3,N, start
```

```
a(0:N-1)=b(0:N-1)+c(0:N-1)
```

```
    li vl,N
    lv vr0,[b]
    lv vr1,[c]
    vadd vr2,vr1,vr0
    sv vr2,[a]
```

Vector Architectures

- General Vector Architecture



- Memory system is wide, and also supports at least strides, and possibly indexed accesses
- The number of lanes can vary up to MVL
- As architectures evolved, vector elements could vary in size (b,h,w,l)
- Mask register evolved into a register file.

Mask & VL Registers

- What mask registers are for:

```
for(x=0;x<N;x++){
  if(c[x]!=0) a[x] = b[x] / c[x];
}
```

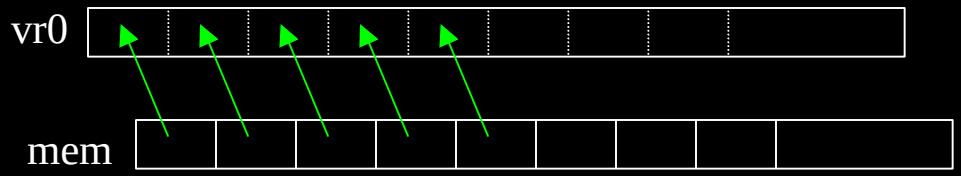
```
lv vr1,[b]
lv vr2,[c]
vcmp.neq mask,vr2,0
vdiv vr0,vr1,vr2
sv vr0,[a]
```

- VL register: length of an array doesn't have to match the hardware MVL, so VL can impose a limit for the case where the length of the array<MVL

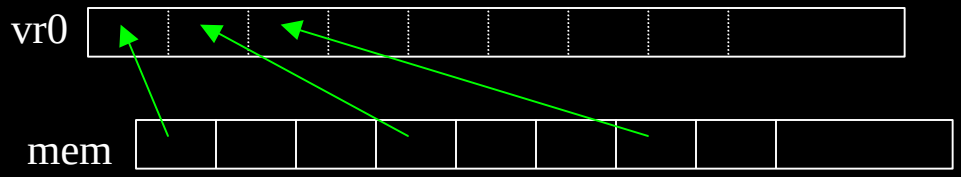
  e.g. array length = 10, MVL = 64

Memory Access

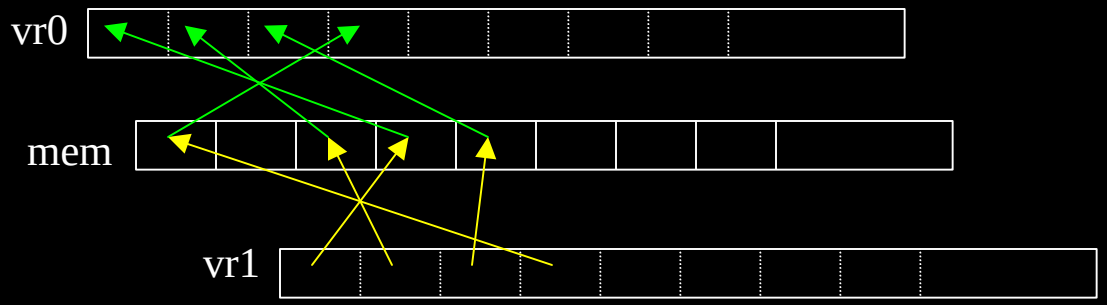
- Unit Stride : access every word



- Non-unit Stride : access every nth word

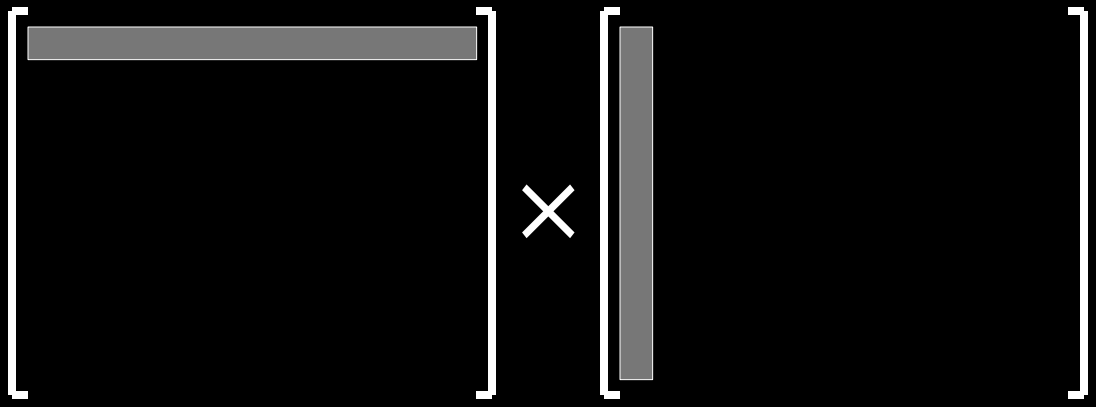


- Indexed : use one vector register as addresses to load into another vector register



Why non unit strides?

- Simple case consider matrix multiply e.g. multiplication involved in dot product, before reduction.
- First vector could be accessed with unit stride, but second must be accessed with non-unit stride



Data Dependence Analysis

- Vectorized loops will execute in parallel. So all dependencies must be checked and ensure that the new code preserves order.

```
for(x=1;x<=N;x++){
  a[x] = …
  … = a[x-1]
}
```

vectorizable as:

```
A(1:N) = …
… = A(0:N-1)
```

```
for(x=1;x<=N;x++){
  a[x-1] = …
  … = a[x]
}
```

not vectorizable as:

```
A(0:N-1) = …
… = A(1:N)
```

vectorizable as:

```
… = A(1:N)
A(0:N-1) = …
```

- Construct dependence graph – see Muchnick ch9

FORTRAN Compiler Evolution

- Initially FORTRAN had vector notation added to it so the compiler could easily recognize it, and exploit the vector architecture.

  e.g. When coding use:

  ```
  A(1:N) = B(1:N) + C(1:N)
  ```

  instead of:

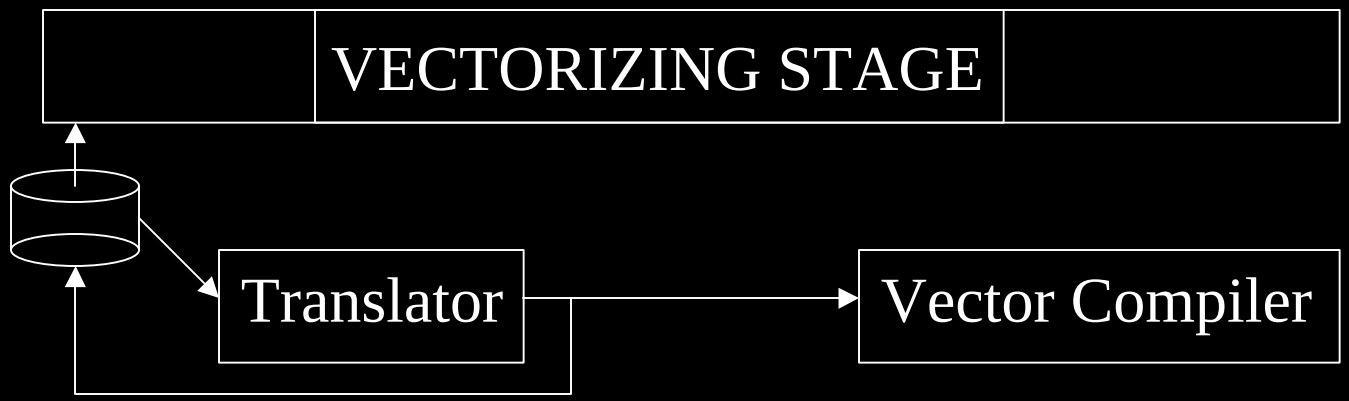
  ```
  DO i=1,N
    A(i) = B(i) +C(i)
  ENDDO
  ```

FORTRAN Compiler Evolution (2)

- But the question is what to do with older programs? - One solution was to automate conversion to the new standard and save the new code. RICE PFC (Parallel FORTRAN Converter)

```
do 10 i=1,N
  A(i) = B(i) + C(i)
10 continue
```

```
A(1:N) = B(1:N) + C(1:N)
```



FORTRAN Compiler Evolution (3)

- Surprisingly this translator was about 10 times faster than the vectorizing compiler they used for comparison.
- This includes all the time necessary for uncovering all the parallelism, and inserting all the bounds checks.

FORTRAN Compiler Evolution (4)

- At this point its just as easy to create a translator as to incorporate the logic into the compiler. (The only difference is speed for multiple compiles)

- Compiling C (which doesn't have the vector notation, doesn't have a simple do loop, and also doesn't have such a strict memory access model) requires much more analysis.

Loop Transformations

- These transformations are preformed on a high level representation to exploit memory access times, separate vectorizable from non-vectorizable code, etc...
- All transformations assume the necessary data dependence analysis has been done.

Strip Mining

- Vector machines have MVL imposed by hardware, so transform loops to match this

```
for(x=0;x<10000;x++){
  a[x] = b[x] + c[x]
}
```

```
for(x=0;x<10000;x+=MVL){
  for(y=x;y<min(x+MVL,10000)){
    a[y] = b[y] + c[y]
  }
}
```

Scalar Expansion

- Here a scalar inside a loop is replaced by a vector so that the loop can be vectorized

```
for(x=0;x<N;x++){
  t = a[x] + b[x];
  z[x] = t / c[x];
}
```

```
for(x=0;x<N;x++){
  t[x] = a[x] + b[x];
  z[x] = t[x] / c[x];
}
```

- Or something like

```
for(x=0;x<N;x++){
  t = a[x] + b[x];
}
```

```
for(x=0;x<N;x++){
  t[x] = a[x] + b[x];
}
```

Cycle Shrinking

- Loop can't be executed completely in parallel, but certain iterations can be. Similar to strip mining.

```
DO i=1,n
  a[i+k] = b[i]
  b[i+k] = a[i] +c[i]
ENDDO
```

```
DO TI=1,n,k
  DO i=TI,TI+k-1
    a[i+k] = b[i]
    b[i+k] = a[i] +c[i]
  ENDDO
ENDDO
```

Loop Distribution

- This transformation is used to separate vectorizable from non-vectorizable code.
- There of course is the requirement that the computation not be affected

```
for(x=0;x<N;x++){
  [vectorizable section]
  [non-vectorizable section]
}
```

```
for(x=0;x<N;x++){
  [vectorizable section]
}
for(x=0;x<N;x++){
  [non-vectorizable section]
}
```

- Thus the first loop after the transformation can be vectorized, and the second can be realized with scalar code

Loop Fusion

- This transformation is just the inverse of distribution.
- It can eliminate any redundant temporaries, increase parallelism, improve cache/TLB performance.
- Be careful when fusing loops with different bounds.

Loop Interchange

- Depending on row or column major storage, it might be appropriate or even necessary to interchange the outer and inner loops.
- From VM or cache standpoint it can be critical
- Its conceivable that a vector machine was designed with only unit, or at least very small strides. Which might prevent certain access patterns from being vectorizable.

Loop Reversal

- Reversal – switch order of loop
  (e.g. n downto 1 instead of 1 to n)
- Can then permit other transformations which were prevented by dependencies.

Coalescing

- Coalescing – convert nested loops to single loop
- Can reduce overhead, and improve scheduling

```
DO i=1,n
  DO j=1,m
    a[i,j] = a[i,j]+c
  ENDDO
ENDDO
```

```
DO T=1,n*m
  i=((T-1)/m)*m +1
  j = MOD(T-1,m)+1
  a[i,j] = a[i,j]+c
ENDDO
```

Array Padding

- If stride is a multiple of the number of banks, get bank conflicts.
- Solution is to pad array to avoid this

  Choose smallest pad p s.t. GCD(s+p,B)=1

  e.g. ensure accesses or stride s goto successive banks

  e.g. s=8, B=8, choose to p=1

Compiling C for Vector Machines

- Problems:
  - Use of dynamic memory in arrays
  - Unconstrained for loops as opposed to DO loop
  - Small functions
  - Side effects / embedded operations
- Solutions:
  - careful analysis of for loops for vectorization
  - inlining function calls

Conversion of loops

- Goal: convert for loop to while loop to DO loop
- Issues: actually recognizing DO loops, side effects

```
while(n){
  *a++ = *b++;
  n--;
}
```

```
while(n){
  t1 = a;
  a=t1+4;
  t2 = b;
  b=t2+4;
  *t1=*t2;
  t3=n;
  n=t3-1;
}
```

```
DO i=0,n-1
  *(a+4*i) = *(b+4*i);
ENDDO
```

Inlining

- Goal: inline small function calls for vectorization

```
for(x=0;x<N;x++){
  A[x] = foo1( B[x] );
}
```

```
A(0:N-1) = foo1_v( B(0:N-1) );
```

- What if it's a library call? - either need to save IR of routine, or create vectorized version of library functions.

Inlining (2)

- What if argument arrays could be aliased? – insert pragma, assume different semantics, or carefully inline it.

```
foo2(int *x,int *y, int *z,int n){
  for(;n;n--)
   *x++ = (*y++) * (*z++);
}
```

```
DO i=0,N-1
  *(&x+4*i)=*(&y+4*i)* *(&z+4*i);
ENDDO
```

```
x(0:N-1) = y(0:N-1) * z(0:N-1)
```

Other issues in C

- Typically large matrices will be built from dynamically allocated sub blocks. As long as the blocks are sufficiently large, vector operations can still be preformed.

Scheduling

- Scheduling is extended from scalar instruction concepts. Still have to include dependence analysis and architectural information (e.g. functional units, ld/st units, latencies, etc…)

Wrap Up

- Speed up is limited by what can not be vectorized,
- some programs can have parallelism of up to 50
- Data dependence analysis can be performed with little impact on compile time.
- Some loop transformations require architectural knowledge

Suggested Reading

- Advanced Compiler Optimizations for Supercomputers - Padua
- Muchnick Chapter 9 for Data Dependence Analysis