REGIONALES HOCHSCHUL-
RECHENZENTRUM KAISERSLAUTERN

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

RHRK-Tutorial

Vectorization: The good, the bad and the ugly

Course instructor: Gabriele Monaco
In charge: Dr. Josef Schüle, RHRK

A simple vectorized loop

```
//main.c

double a[MAX], b[MAX], c[MAX];

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

```
//main_novec.s

vmovsd    -120(%rsp,%rax), %xmm0
vaddsd    255880(%rsp,%rax), %xmm0, %xmm0
vmovsd    %xmm0, 511880(%rsp,%rax)
addq      $8, %rax
```

```
//main_vec.s

vmovapd   511880(%rsp,%rax), %zmm1
vaddpd    255880(%rsp,%rax), %zmm1, %zmm0
vmovapd   %zmm0, -120(%rsp,%rax)
addq      $64, %rax
```

Compile with the -S flag to see the assembly code
Here special packed double precision (pd) instructions are used on the avx512 registers, the non vectorized code looks very similar, but smaller registers are used and the instructions are scalar double precisions (sd)

Auto vectorization

- **If instructed to do so, the compiler is capable of vectorizing loops, but the code needs to be vectorizable on the first place!**
- **In some cases vectorization would produce wrong results (e.g. data dependencies)**
- **In others it may not be worth it (stride much bigger than 1)**

- **Good**
  - **Simple assignments**
    - a[i] = 0
  - **Vector statements**
    - c[i] = a[i] + b[i]
- **Bad**
  - **Function calls**
    - c[i] = compute(a[i], b[i])
  - **Data dependencies**
    - c[i] = a[i] + c[i-1]
  - ***And so on..***

Additional examples

Good

- Loops with clear limits and use the loop counter

```
for(i=0;i<N;i++)

for(i=0;i<N;i++)
  for(j=0;j<i;j++)
```

- Avoid branches inside the loop

```
if(condition)
  for(i=0;i<N;i++)
    code_condition_1
else
  for(i=0;i<N;i++)
    code_condition_2
```

- Avoid using pointers instead of plain arrays

```
float *mat;
for(i=0;i<rows;i++)
  for(j=0;j<cols;j++)
    mat[(i)*cols+j]=mat[(i-1)*cols+j]+..
```

Bad

```
while(i)

for(;;) { if(condition) break; }

for(i=0;i<N;)
  i++
```

```
for(i=0;i<N;i++)
  if(condition)
    code_condition_1
  else
    code_condition_2
```

```
float **mat;
for(i=0;i<rows;i++)
  for(j=0;j<cols;j++)
    mat[i][j]=mat[i-1][j]+..
//mat[i] and mat[i-1] may be the same
```

Additional examples (cont'd)

Good **Bad**

- Try to make contiguous accesses (favour unit stride in loops)

```
for(i=0;i<N-1;i+=2) {
    x[i]=1.0f0;
    x[i+1]=-1.0f0;
}
```

```
for(i=0;i<N;i+=2)
    x[i]=1.;
for(i=1;i<N;i+=2)
    x[i]=-1.;

for(i=0;i<N;i++) y[i]=x[ind[i]];
```

- Use structure of arrays (SoA) instead of array of structures (AoS)

```
struct SOA {
    double x[MAX],y[MAX];
    char c[MAX];
}
SOA soa;
for(i=0;i<MAX;i+)
    SOA.x[i]=0.0f0;
//Stride 1 access in memory
```

```
struct AOS {
    double x,y;
    char c;
}
AOS aos[MAX];
for(i=0;i<N;i++)
    AOS[i].x=0.0f0;
//Stride 3 access in memory
```

- Align data accesses and structures (more on this later..)

Main obstacles for vectorization

- **Function calls: the compiler doesn't know how to execute general functions on a vector, however that may not be true on some well known functions in the math library (`sin()`, `log()`, `fmax()` ...) or for inline functions.**
- **Non contiguous memory access: when the loop stride is larger than 1, memory loads are less efficient, as contiguous reads are usually faster and this may prevent the loop from being vectorized (although physically possible)**
- **Data dependencies: as vectorization alters the order of the instructions (introducing parallelism), the vectorized code must produce the same result as the scalar one, hence instruction that may be executed simultaneously must not depend on each other**
  - **Read-after-write: A[j]=A[j-1]+1**
  - **Write-after-read: A[j-1]=A[j]+1**

Pointer aliasing

- When dealing with pointers (rather than static arrays), the compiler may not know where each location is actually pointing
- It may think that two pointers are overlapping and can bring data dependencies
- Sometimes we know that pointers are not overlapping, the compiler needs to be informed to be able to vectorize

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

int main() {
    int a[N], b[N], c[N];
    //we know the loop is safe to
vectorize..
    func(N, a, b, c);
}
```

Compilation hints

- The C standard supports giving additional informations to the compiler by using the pragma preprocessor directives from within the code (usually compiler dependend)
- Tell the compiler a loop is safe to vectorize
  - `#pragma ivdep //intel compiler`
  - `#pragma GCC ivdep //gnu compiler`
- Tell the compiler to vectorize a loop (even if not worth it)
  - `#pragma vector always //intel compiler only`
- Tell the compiler not to vectorize the loop
  - `#pragma novector //intel compiler only`
- Inform the compiler about the expected length of the loop
  - `#pragma loop_count(n) //intel compiler only`

The `restrict` keyword

- This qualifier is used to indicate a pointer whose referenced memory is not aliased (not other pointers are referencing the same piece of memory in a way the compiler doesn't expect)
- As a result the compiler can produce much cleaner code can optimize more
- The function from the previous example can be vectorized!

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

- **High Performance Computing on Elwetritsch**

Vielen Dank
Thank You