CS528
SIMD/GCC/FP

A Sahu
Dept of CSE, IIT Guwahati

Outline

- Intro to Code Optimization
- Machine independent/dependent optimization
- Common sense of Optimization
  - Do less work, avoid expensive Ops, shrink working set
- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization
- Scalar Profiling
    - Manual Instrumentation (get_wall_time, clock_t)
    - Function and line based profiling (gprof, gcov)
    - Memory Profiling (valgrind, callgraph)
    - Hardware Performance Counter (oprofile,likwid)

Further Optimizations for Serial Code

- Simple measure Large impact : simd, branch, comm sub expre
- C++ Optimization

Simple measures, large impact

- Elimination of Common Sub-expressions
- Avoid Branches:
  - **Code Can be SIMdized by compiler/gcc**
  - **Effective use of pipeline for loop code**
- Use of SIMD Instruction sets
  - **512 bit AVX SIMD in modern processor**
  - **ML/AI app use 8 bit Ops, can be speed up 512/8=64 time by simply SIMD-AVX**

Elimination of Common Sub-expressions

```
//value of s, r, x don't change in this loop
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+s+r+sinx(x);
}
```

```
//value of s, r, x don't change in this loop
Tmp=s+r+sinx(x);
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+Tmp;
}
```

Avoid Branches

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

```
for (i=0; i<N; i++){
     for(j=0;j<i;j++)
         C[i] =C[i] -A[i][j]*B[i];
     for(j=i;j<N;j++)
         C[i] =C[i] +A[i][j]*B[i];
}
```

Use of SIMD: independent loop iteration

```
for (i=0; i<ALargeN; i++){
    A[i]=A[i]+B[i]*D[i];
}
All iterations in this loop are
independent : gcc SIMD utilize very
nicely
//ML application uses 8 bit OPS, 512
bit AVX SIMD 512/8=64 OPS can be done
in  parallel.
```

The ith iteration access : A[i], B[i], D[i]

Use of SIMD: independent loop iteration

$$S = \sum_{i=0}^{N} w_i x_i$$

- Vector dot product : is the most ***common and frequent kernel*** in
  - Matrix multiplication,
  - Neuron calculation (neural network NN)
    - Conv NN, Deep NN, //ML domain
  - Digital Signal Processing, Image Sig Processing, etc
  - Media Applications : audio, video, JPG/MPG, DCT.

Use of SIMD: independent loop iteration

$$S = \sum_{i=0}^{N} w_i x_i$$

```
for (i=0; i<ALargeN; i++){
    S=S+A[i]+B[i];
}
All iterations in this loop are
independent : gcc SIMD utilize very
nicely
//ML application uses 8 bit OPS, 512 bit
AVX SIMD 512/8=64 OPS can be done in
parallel.
```

The ith iteration access : A[i], B[i]

Use of SIMD: independent loop iteration

```
for (i=0; i<N; i++){
    A[i]=A[i]+B[i]; //S1
    B[i+1]=C[i]+D[i]; //S2
}
```

The ith iteration access : A[i], B[i], B[i+1],C[i] D[i]

Dependent loop iteration : i and i+1

Use of SIMD: independent loop iteration

```
A[0]=A[0]+B[0];
for (i=0; i<N; i++){
    B[i+1]=C[i]+D[i]; //S2
    A[i+1]=A[i+1]+B[i+1];//S1
}
B[N]=C[N-1]+D[N-1];
```

The ith iteration access :
A[i+1], B[i+1],C[i], D[i]

Use of SIMD: independent loop iteration

```
for (i=0; i<N; i++){
    A[i]=A[i]+B[i]; //S1
    B[i+1]=C[i]+D[i]; //S2
}
```

```
A[0]=A[0]+B[0];
for (i=0; i<N; i++){
    B[i+1]=C[i]+D[i]; //S2
    A[i+1]=A[i+1]+B[i+1];//S1
}
B[N]=C[N-1]+D[N-1];
```

Use of SIMD: independent loop iteration

- Affine access : index a.x+b form

```
for (i=0; i<N; i++){
    X[a*i+b]=X[c*i+d];
    //where a,b,c,d are integer
}
```

- GCD(c,a) divides (d-b) for loop dependence
- Ref Book : Hennesy Paterson, Advanced Computer Architecture, 5th Edition Book,

GCD test Example

```
for (i=0; i<N; i++){
        X[2*i+3]=X[2*i]+5.0;
      //X[a*i+b]=X[c*i+d];
}
```

- GCD(c,a) must divides (d-b) for loop dependence
- Value of a=2, b=3, c=2, d=0;
- GCD(a,c)=2, d-b=-3
- 2 does not divide -3 → No dependence Possible

Role of Compilers

- General Compiler Optimization Options
- Inlining
- Aliasing
- Computational Accuracy

General Compiler Optimization Options

- GCC optimization : -O0, -O1, -O2,-O3
- $man gcc
- At –O0 level:
  - Compiler refrain from most of the opt.
  - It is correct choice for analyzing the code with debugger
- At high level
  - Mixed up source lines, eliminate redundant variable, rearrange arithmetic expressions
  - Debugger has a hard time to give user a consistent view on code and data

General Compiler Optimization Options

- Level 1
  - fauto-inc-dec, -fmove-loop-invarient, -fmerge-constants, -ftree-copy-prop, -finline-fun-called-once
- Level 2
  - -falign-functions, -falign-loops, level, -finlining-small-fun, -finling-indirect-fun, -freorder-fun, -fstrict-aliasing
- Level 3
  - -ftree-slp-vectorize, -fvect-cost-model

Inlining

- Inlining
  - Tries to save overhead by inserting the complete code of function
  - At the place where it called
- Saves time and resources by
  - not using function call, stack
  - All complier to use registers
  - Allows compiler to views a larger portion of code and employ OPTimization
- Auto inline or hint in program to function to be inlined

Aliasing

```
void scale_shift(double *a, double *b,
                 double s, int n){
 for(int i=1;i<n;i++)
   a[i]=s*b[i-1];
}
```

- Assuming a and b don’t overlap
  - **double __restrict *a, double __restrict *b**
  - **__restrict say no overlap**
- Load and stores in the loop can be rearranged by compiler
- Apply software-pipeling, unrollling, group load/store, SIMD, etc

Computational Accuracy

- Compiler some time refrain from re-arranging arithmetic expression
- FP domain associative rule $a+(b+c) \neq (a+b)+c$
- If accuracy need to be maintained
  - Compared to non optimized code
  - Associative rules must not be used by compiler
  - Should be left to programmer to regroup safely
- FP underflow are push to zero

Computational Accuracy

- FP domain associative rule **$a+(b+c) \neq (a+b)+c$**
  - Let $a=1.0\text{x}10^{38}$, $b=-1.0\text{x}10^{38}$, $c=1$
  - Result of $a+(b+c)$

    $= 1.0\text{x}10^{38} + (-1.0\text{x}10^{38} +1)$

    $=1.0\text{x}10^{38} + (-1.0\text{x}10^{38}) = 0$ **//Big+Small=Big**
  - Result of $(a+b)+c$

    $= (1.0\text{x}10^{38} + -1.0\text{x}10^{38})+1$

    $=0+1 \quad = 1$

Computational Accuracy

- Why it happens for FP?
  - **FP format use 32 bit represent number up to $\pm 2^{127}$**
    - **Int use 32 bit represent up to $\pm 2^{31}$**
  - Used same 32 bit for large numbers, numbers are not equal-spaced
  - From 36000ft, both IITG and Amingoan are not distinguishable [Resolution:]
  - Going by Air: Delhi, Noida, Gurgaon use the same Airport