CS528
High Performance Computing

SCO
and
Data Access Optimization

A Sahu

Dept of CSE, IIT Guwahati

Outline

- SCO:
  - Simple measure Large impact : simd
  - Role of Compiler
    - Aliasing, Accuracy
  - C++ Optimization
- Data Access Optimization
  - Roofline Model
  - Caching optimization
  - App classification based DA: N/N, N^2/N^2, N^3/N^2
- ***[Ref: Hager Book, PDF uploaded to MS Team]***

Simple measures, large impact

- 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**

Avoid Branches: can be simidized

```
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

```
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
  - **One can refer any Adv. Compiler 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]+k;
}
```

- 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 $(a+b)+c$

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

    $= 0+1 = 1$

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

C++ Optimizations

- Temporaries
- Dynamic Memory Management
- Loop Kernel and Iterators

C++ Opt: Temporaries

- C++: operator overloading uses

```
class vec3d{
        double x,y,z;
public: vec3d( double _x=0.0, _y=0.0, _z=0.0):x(_x),y(-y),z(_z){}
        vec3d operator+(const vect3d &oth){
                vec3d tmp;  tmp.x=x+oth.x; …for y, and z
                return tmp
        }
        vec3d operator*(double s, const vec3d &v){
        vec3d tmp(s*v.x, s*v.y,s*v.z); return tmp;}
}
main() {
        vec3d   a,   b(2,2),  c(3);   double u=1.0,v=2.0;
        a=u*b + v*c;
}
```

C++ Opt: Temporaries

- C++: operator overloading uses
- In this prev statements
  - Constructor get called for a,b,c
  - Operator*, constructor for tmp, destructor for tmp
  - Operator*, constructor for tmp, destructor for tmp
  - Operator+, constructor for tmp, destructor for tmp
  - Copy constrtor called with tmp
- **Simply we could have write**
  - **a.x=u*b.x+v*c.x; a.y=u*b.y+v*c.y; a.z=u*b.z+v*c.z;**

C++ Opt: Dynamic Memory Management

```
void func(double Th, int Len){
 vector<double> v(Len);
 if(rand()>Th*RAND_MAX){
     v=obtain_data(Len);
     sort(v.begin(),v.end());
     process_data(v);
     }
}
```

This creation is Costly

C++ Opt: Dynamic Memory Management

```
void func(double Th, int Len){

  if(rand()>Th*RAND_MAX){
      vector<double> v(Len);
      v=obtain_data(Len);
      sort(v.begin(),v.end());
      process_data(v);
      }
}
```

This creation is Costly, so make it Lazy

- Lazy construction : if the probability of requirement is low
  - Post pone the construction if the condition become true

C++ Opt: Dynamic Memory Management

```
void func(double Th, int Len){
 static vector<double> v(LargeLen);
 if(rand()>Th*RAND_MAX){
     v=obtain_data(Len);
     sort(v.begin(),v.end());
     process_data(v);
     }
}
```

One time construction for all calls

- Static Construction : if the probability of requirement is high or always required
  - one time Construction : for all call/invocation
  - Take sufficient largeLen

C++ Opt: Loop Kernel and Iterators

- Runtime of scientific application dominated by loops or loops nest
- Compiler ability to optimize loops is pivotal for getting performance
- Operator overloading and template may hinders good loop optimization

C++ Opt: Loop Kernel and Iterators

- Non-SIMDized code: operator[] called twice for a and b, compiler refuse to SIMDize

```
template<class T>
T Sprod(cosnt vector<T> &A,
        const vector<T> &B){

    T result=T(0);
    int s=A.size();
    for(int i=0;i<s;i++)
        result += A[i]*B[i]; //Access
    return result;
}
```

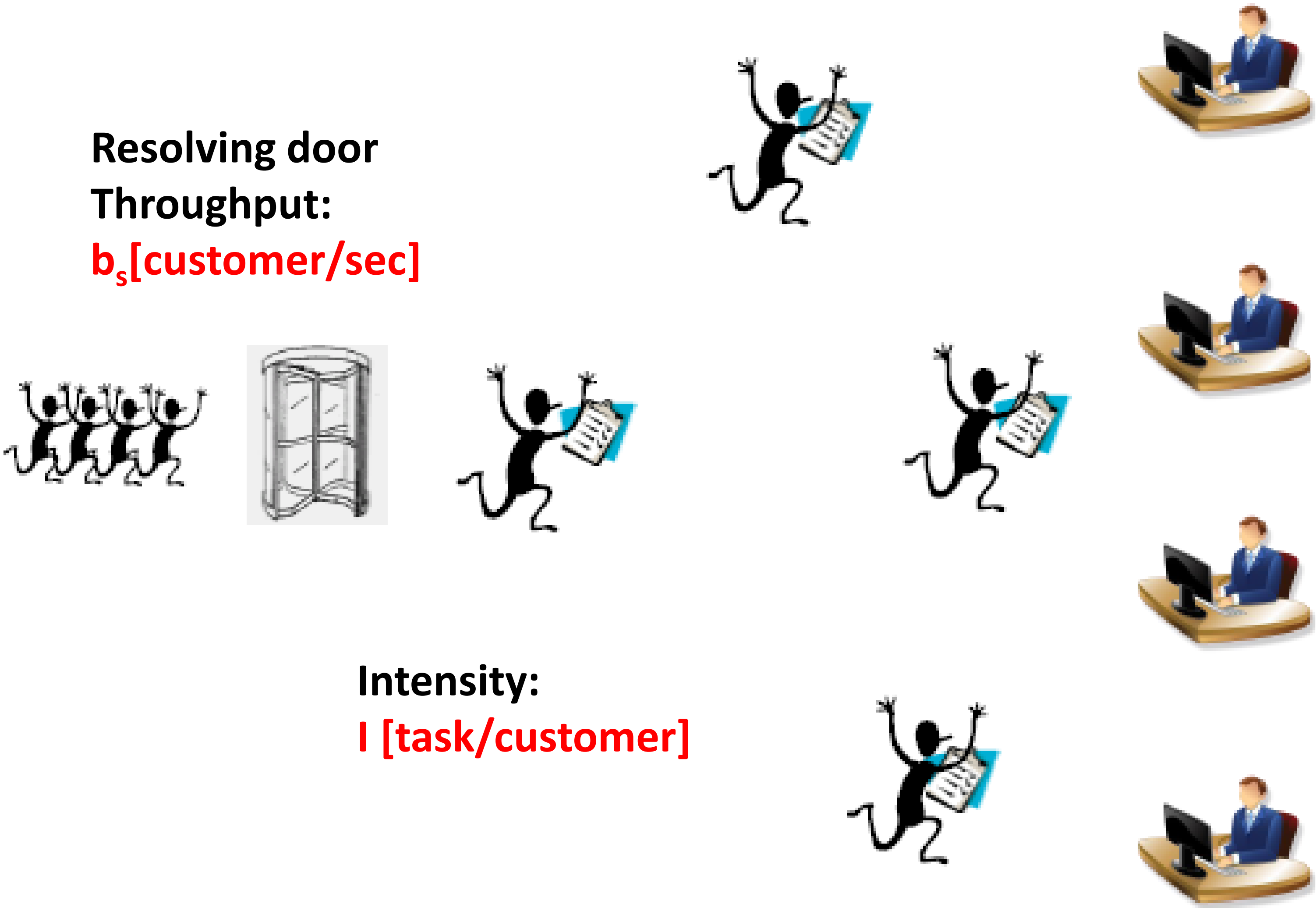
C++ Opt: Loop Kernel and Iterators

- SIMDized

```
template<class T>
T Sprod(cosnt vector<T> &A,
        const vector<T> &B){
vector<T>::const_iterator
        iA=A.begin(), iB=B.begin();
    T result=T(0);
    int s=A.size();
    for(int i=0;i<s;i++)
       result += iA[i]*iB[i];//Access
    return result;
}
```

Data Access Optimization

Performance of System: Modeling Customer Dispatch in a Bank



Resolving door Throughput:
b_s[customer/sec]

Processing Capabilty:
P_{peak} [task/sec]

Intensity:
I [task/customer]

Modeling Customer Dispatch in a Bank

- How fast can tasks be processed? $\boldsymbol{P}$**[tasks/sec]**
- The bottleneck is either
  - The service desks (peak. tasks/sec): P_{peak}
  - The revolving door (max. customers/sec): $I \cdot b_S$
- Performance $P = \min(P_{\text{peak}},\ I \cdot b_S)$
- **This is the "Roofline Model"**
  - High intensity: P limited by "execution"
  - Low intensity: P limited by "bottleneck"