CS528
C++ opt
&
Data Access Optimization

A Sahu

Dept of CSE, IIT Guwahati

Outline

- C++ Optimization
- **Data Access Optimization**
- **Roof line Model**

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

DAO

- 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 Website]***

Performance of System: Modeling Customer Dispatch in a Bank

Resolving door Throughput:
b_s[customer/sec]

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

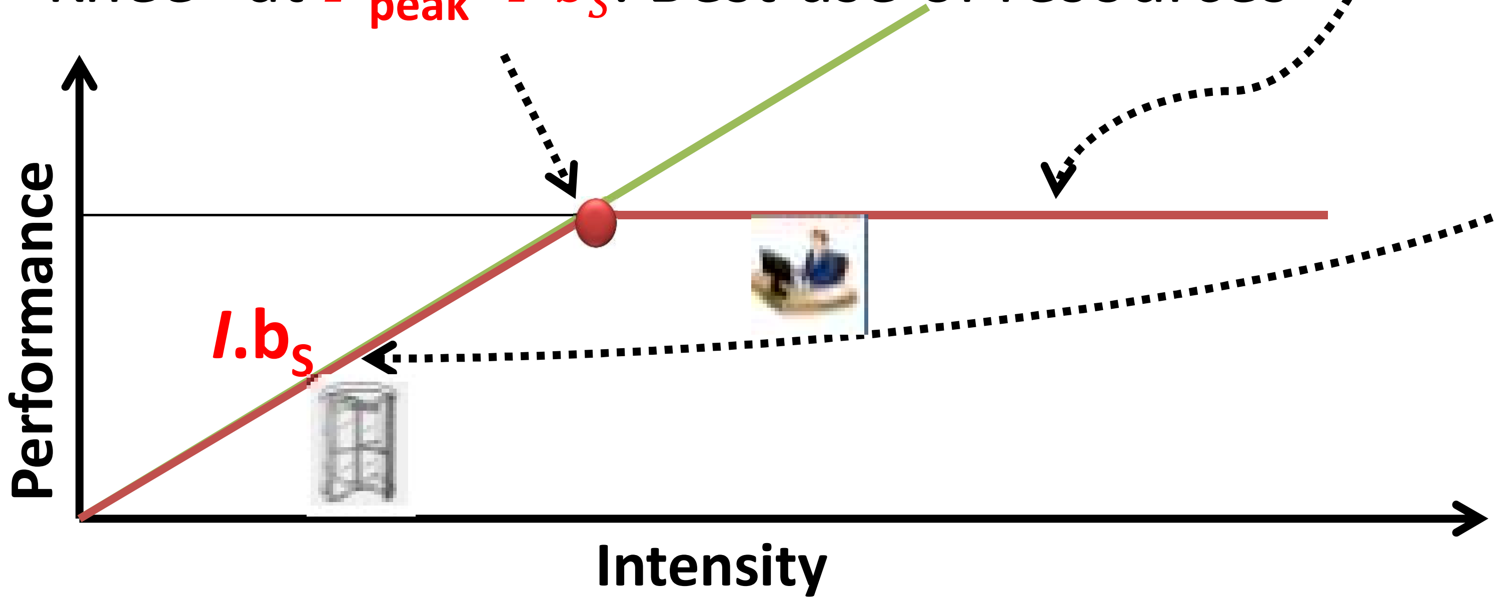
Intensity:
I [task/customer]

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): $P_{\mathbf{peak}}$
  - The revolving door (max. customers/sec): $I \cdot b_S$
- Performance P=min($P_{\mathbf{peak}}$, $I \cdot b_S$)
- **This is the "Roofline Model"**
  - High intensity: P limited by "execution"
  - Low intensity: P limited by "bottleneck"

Modeling Customer Dispatch in a Bank

- Performance $P=\min(P_{peak}, I{\cdot}b_S)$
- **This is the "Roofline Model"**
  - High intensity: P limited by "execution"
  - Low intensity: P limited by "bottleneck"
  - "Knee" at $P_{peak}=I{\cdot}b_S$: Best use of resources



- Roofline is an "optimistic" model

The Roofline Model

- P_{max} = Peak performance of the machine
- I = Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck")
- b_S = Applicable peak bandwidth of the slowest data path utilized

Expected performance:

$$P = \min(P_{\text{peak}},\ \overset{[F/B]}{I} \cdot \overset{[B/s]}{b_S})$$

Apply Roof line to Machine and Code

- Machine Parameter 1 : P_{peak} [F/s]=4 G F/S
- Machine Parameter 2 : b_S [B/s] =10 G B/s
- Application Properties: I [F/B] = 2F/8B=0.25F/B

for(i=0;i<N;i++) s=s+a[i]*a[i]; // double s, a[]

- Performance = P = min(P_{peak}, I*b_s)

=min(4 GF/s, 0.25 F/B *10 G.B/s)

=min(4 GF/s, 2.5 GF/s)

=2.5 G F/s

The Refine Roofline Model

- P_{max}= Peak performance of a loop assuming that data comes from L1 cache **(is not necessarily P_{peak})**
- I= Computational intensity ("work" per byte transferred) over the slowest data path utilized ("the bottleneck"),
- **Code Balance B_c=I^{-1}= in Byte per Flop**
- b_S= Applicable peak bandwidth of the slowest data path utilized

Expected performance:

$$P = \min(P_{\text{max}}, \overset{[F/B]}{I} \cdot \overset{[B/s]}{b_S})$$

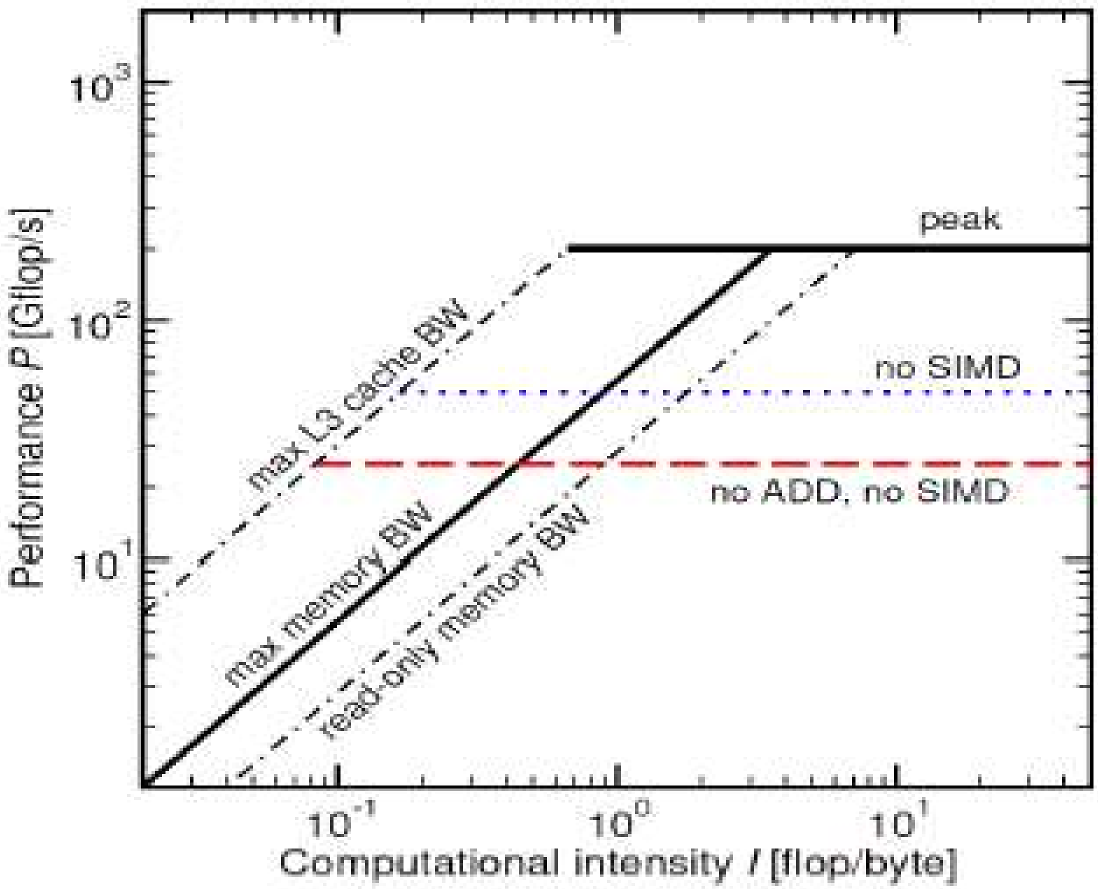
$$P = \min(P_{\text{max}}, b_S / B_c)$$

Factors to consider in Roofline Model

- BW Bound : may be simple
  - Accurate traffic calculation (write allocate, stride,..)
  - Practical not equal to theoretical BW limits
  - Saturation effects -> consider full socket only
- Core Bound : may be complex
  - Multiple bottlenecks: LD/ST, arithmetic, pipeline, SIMD, execution port
  - Limit is linear in # of cores

Refine RFL model: Graphical Representation

- Multiple ceiling may apply
  - Diff BW/Data paths -> Diff inclined ceilings
  - Different P_{max}-> Diff flat ceilings



P_{max} comes from code analysis : with/without SIMD, add other FUs

Apply Roof line to Haswell Core to Triad Code

- Achievable Max Performance 1: P_{max} [F/s]=12.27 G F/S
- Machine Parameter 2 : b_S [B/s] =50 G B/s
- Application Properties: I [F/B] = 2F/40B=0.05F/B

```
for(i=0;i<N;i++) a[i]=b[i]+c[i]*d[i]; // double a,b,c,d
```

- Performance = P = min(P_{max}, I*b_s)

  =min(12.27 GF/s, 0.05 F/B *50 G.B/s)

  =min(12.27 GF/s, 2.5 GF/s)

  =2.5 G F/s

Code Balance/Intensity Examples

```
for(i=0;i<N;i++)//Copy
    a[i]=b[i];
```

B_c=24B/0F=NA

```
for(i=0;i<N;i++)//Scale
      a[i]=s*b[i];
```

B_c=24B/1F=24 B/F

```
for(i=0;i<N;i++) //Add
      a[i]= b[i]*c[i];
```

B_c=32B/1F=32 B/F

A[i]=B[i];//Require reading of A[i], B[i] and writing to A[i] incase of L1 cache access; so 2 read and one write

Code Balance/Intensity Examples

```
double a[N],b[N],C[N], d[N];
for(i=0;i<N;i++)
   a[i]=a[i]+b[i];
```

B_c=24B/1F=24 B/F
I=0.042F/B

```
for(i=0;i<N;i++) //Triad
      a[i]=a[i]+s*b[i];
```

B_c=24B/2F=12 B/F
I=0.083 F/B

```
for(i=0;i<N;i++) //float a[]
      s= s+a[i]*a[i];
```

B_c=4B/2F=2 B/F
I=0.5 F/B

```
for(i=0;i<N;i++)//float a[],b[]
      s= s+a[i]*b[i];
```

B_c=8B/2F=4 B/F
I=0.25 F/B