DPHPC: Balance Principle, SIMD
Recitation session, 28.11.2019
Deriving a balance principle

- **Concept of balance**: a computation running on some machine is efficient if the compute-time dominates the I/O time. [Kung, 1986]

- **Deriving a balance principle**:
  - Algorithmically analyze the parallelism
  - Algorithmically analyze the I/O behavior (i.e., number of memory transfers)
  - Combine these two analyses with a cost model for an abstract machine

- **Goal**: say precisely and analytically how
  - Changes to the architecture might affect the scaling of a computation
  - Identify what classes of computation might execute efficiently on a given architecture

Analyzing I/Os

- We use the classical external memory model
- Two level memory
  - One large & slow
  - The other small & fast (capacity: $Z$ words)
    *It can be an automatic cache or a software-controlled scratchpad*
- Work operations can be performed only on data in fast memory
- Slow $\leftrightarrow$ Fast memory transfers occur in blocks of $L$ words
- $Q_{z,L}(n)$ is the number of $L$-sized transfers between slow and fast memory for an input of size $n$

Goal is to optimize the computational intensity: $\frac{W(n)}{Q_{z,L}(n) \times L}$

Architecture-Specific Cost Model

- We need to introduce the time
  - This depends on the specific architecture
- \( p \) cores
- Each core can deliver \( C_0 \) operations per time
- The time to transfer \( m \cdot L \) words is:
  - \( \alpha + m \times L/\beta \)
  - \( \alpha \) is the latency
  - \( \beta \) is the bandwidth in units of words per time

- The best possible compute time is (Brent’s theorem):

\[
T_{\text{comp}}(n; p, C_0) = \left(D(n) + \frac{W(n)}{p}\right) \times \frac{1}{C_0}
\]
Architecture-Specific Cost Model

- \( Q_{Z,L}(n) \) is for the sequential case
- We need to move to the parallel case \( Q_{p;Z,L}(n) \)
  - We can bound \( Q_{p;Z,L}(n) \) in terms of \( Q_{Z,L}(n) \)
  - Need to select a specific scheduler!
  - Compute it directly
- Assumptions:
  - the latency is accounted for each node in the critical path
  - all the \( Q_{p;Z,L}(n) \) are aggregated and pipelined by the memory system
  - Hence they are delivered at the peak bandwidth

- We can estimate the memory cost as:
  \[ T_{mem}(n; p, Z, L, \alpha, \beta) = \alpha \times D(n) + \frac{Q_{p;Z,L}(n) \times L}{\beta} \]

Blelloch et al., „Low depth cache-oblivious algorithms.” SPAA 2010
The Balance Principle

- The balance principle follows by imposing $T_{mem} \leq T_{comp}$

$$T_{mem}(n; p, Z, L, \alpha, \beta) = \alpha \times D(n) + \frac{Q_p; Z; L(n) \times L}{\beta}$$ $T_{comp}(n; p, C_0) = \left( D(n) + \frac{W(n)}{p} \right) \times \frac{1}{C_0}$

The Balance Principle: matrix-matrix product

- Clearly, $W(n) > D(n)$ and

$$Q_{p;Z,L}(n) \geq \frac{W(n)}{\sqrt{2 \times L \times \sqrt{Z/p}}}$$

- We get the following

$$\frac{p \times C_0}{\beta} \leq \frac{Z}{\sqrt{p}}$$

- What if we double the number of cores?

Flynn’s Taxonomy

Flynn’s Taxonomy

Vectors, GPUs

Flynn’s Taxonomy

Vectors, GPUs  Multiprocessing

Libraries

**Using Vec3D = std::array<float, 3>;**

```cpp
float scalar_product(Vec3D a, Vec3D b) {
}
```

**Using Vec3D = std::array<_m128, 3>;**

```cpp
__m128 scalar_product(Vec3D a, Vec3D b) {
    return _mm_add_ps(
        _mm_add_ps(
            _mm_mul_ps(a[0], b[0]),
            _mm_mul_ps(a[1], b[1]))
        ,
        _mm_mul_ps(a[2], b[2]))
    ;
}
```
Libraries

using Vec3D = std::array<float, 3>;
float scalar_product(Vec3D a, Vec3D b) {
}

using Vc::float_v;
using Vec3D = std::array<float_v, 3>;
float_v scalar_product(Vec3D a, Vec3D b) {
}

using Vec3D = std::array<__m128, 3>;
__m128 scalar_product(Vec3D a, Vec3D b) {
    return _mm_add_ps(_mm_add_ps(_mm_mul_ps(a[0], b[0]), _mm_mul_ps(a[1], b[1])), _mm_mul_ps(a[2], b[2]));
}
Libraries

using Vec3D = std::array<float, 3>;
float scalar_product(Vec3D a, Vec3D b) {
}

using Vc::float_v;
using Vec3D = std::array<float_v, 3>;
float_v scalar_product(Vec3D a, Vec3D b) {
}

using Vec3D = std::array<_m128, 3>;
__m128 scalar_product(Vec3D a, Vec3D b) {
    return _mm_add_ps(_mm_add_ps(_mm_mul_ps(a[0], b[0]), _mm_mul_ps(a[1], b[1])), _mm_mul_ps(a[2], b[2]));
}

VcDevel/Vc
#include <vector>

void foo( std::vector<unsigned>& lhs, std::vector<unsigned>& rhs ) {
    for( unsigned i = 0; i < lhs.size(); i++ ) {
        lhs[i] = ( rhs[i] + 1 ) >> 1;
    }
}

godbolt.org/z/fTCD_e
Compiler auto-vectorization II - reports

```cpp
#include <vector>

void foo( std::vector<unsigned>& lhs, std::vector<unsigned>& rhs ) {
    for( unsigned i = 0; i < lhs.size(); i++ ) {
        lhs[i] = ( rhs[i] + 1 ) >> 1;
    }
}
```

godbolt.org/z/Sk_8Tq
#include <vector>

```cpp
void foo( std::vector<unsigned>& lhs, std::vector<unsigned>& rhs ) {
    for( unsigned i = 0; i < lhs.size(); i++ ) {
        lhs[i] = ( rhs[i] + 1 ) >> 1;
    }
}
```

godbolt.org/z/yap5-__
Vectorization with dataflow

```c
float f(float a, float b) {
    float r;
    if (a > b) r = a + 1;
    else r = a - 1;
    return r;
}
```

Karrenberg et al., “Whole-Function Vectorization”
Vectorization with dataflow

float f(float a, float b) {
    float r;
    if (a > b) r = a + 1;
    else r = a - 1;
    return r;
}

float f'(float a, float b) {
    bool mask = a > b;
    float s = a + 1;
    float t = a - 1;
    float r = select(mask, s, t);
    return r;
}
float f(float a, float b) {
    float r;
    if (a > b) r = a + 1;
    else r = a - 1;
    return r;
}

float f'(float a, float b) {
    bool mask = a > b;
    float s = a + 1;
    float t = a - 1;
    float r = select(mask, s, t);
    return r;
}

__m128 f_sse(__m128 a, __m128 b) {
    __m128 mask = _mm_cmpgt_ps(a, b);
    __m128 s = _mm_add_ps(a, _mm_one);
    __m128 t = _mm_sub_ps(a, _mm_one);
    __m128 r = _mm_blendv_ps(mask, s, t);
    return r;
}

Karrenberg et al., “Whole-Function Vectorization”
Vectorization pitfalls

```c
void bar(float *A, float* B, float K, int n) {
    for (int i = 0; i < n; ++i)
}
```
Vectorization pitfalls

```c
void bar(float *A, float* B, float K, int n) {
    for (int i = 0; i < n; ++i)
}
```

What if `bar(ptr, ptr + 1, 0.0, 10)`? Runtime check is needed!

https://godbolt.org/z/u2-6Vd
Tips

• Prefer known trip counts (no unpredictable exits)
• Write simple loops (no complex control-flow)
• Branching code? Masking might help.
• Avoid loop-carried dependencies (scan maybe?)
• Use loop index to index arrays and avoid pointer arithmetic
• Prefer consecutive memory access, avoid indirection.
• Check output of compiler and use libraries, flags and directives to help.

Tutorial: https://easypert.net/blog/2017/10/24/Vectorization_part1
Docs: https://llvm.org/docs/Vectorizers.html