Parallel Programming

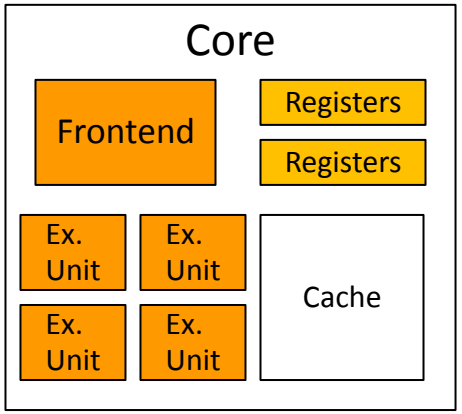
Instruction Level Parallelism

Jan Schönherr

Technische Universität Berlin
School IV – Electrical Engineering and Computer Sciences
Communication and Operating Systems (KBS)
Einsteinufer 17, Sekr. EN6, 10587 Berlin

Instruction Level Parallelism

- Components of a processor core
  - Frontend fetches and decodes instructions
  - Backend schedules instructions to executions units



- How to make use of these components?
  - Hardware: keep all components busy, i. e., provide ILP
    - Pipelining, superscalar processors, out-of-order execution, branch prediction, speculative execution, prefetching, …
  - Software: deliver enough work, i. e., use ILP

Making use of ILP

- A subcategory of “optimizing your program”
- Hardware sees a stream of instructions with associated data
  - Problems arise if data is not available
  - Hardware can bypass/hide these problems only to a certain extent
- Software should provide an “optimized” stream
  - Minimized dependencies among instructions
  - Minimized (unpredictable) jumps
  - Using fast data storage (registers, cache)
- This is your job!
  - (The compiler helps, but it is not almighty .)

Exemplary memory speeds

CORE 0 READ BANDWIDTH IN GB/S

	Exclusive			Modified			RAM
	L1	L2	L3	L1	L2	L3	
Local	45.6	31.1	26.2	45.6	31.1	26.2	10.1
Core1	19.3	19.7		9.4	13.2		
Core4	9.0	9.2		5.6			6.3

CORE 0 WRITE BANDWIDTH IN GB/S

	Exclusive			Modified			RAM
	L1	L2	L3	L1	L2	L3	
Local	45.6	28.8	19.9	45.6	28.8	19.9	8.4
Core1	23.4	22.2	17.6	9.4	13.0		
Core4	9.0			8.3		9.6	5.5

READ LATENCIES OF CORE 0, ALL RESULTS IN NANOSECONDS (CYCLES)

Source	Exclusive cache lines			Modified cache lines			Shared cache lines			RAM
	L1	L2	L3	L1	L2	L3	L1	L2	L3	
Local	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	1.3 (4)	3.4 (10)	13.0 (38)	65.1
Core1 (on die)	22.2 (65)			28.3 (83)	25.5 (75)		13.0 (38)			
Core4 (QPI)	63.4 (186)			102 - 109			58.0 (170)			106.0

- Taken from:

  Molka, D.; Hackenberg, D.; Schöne, R.; Müller, M. S.: *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System.* Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09), IEEE Computer Society, 2009, 261-270

Special case: SMT

- Simultaneous multithreading
  - The ability to execute multiple threads within one core at the same time
  - Without (costly) OS-driven context switches
- Multiple register sets in a core
  - Frontend fetches from multiple locations
  - Backend executes instructions as usual
    - More work -> better utilization
- Software sees TLP, hardware sees ILP
- Trade-off between throughput and latency

Optimization in general

- Why?
  - Specification violated (e. g. time/memory constraints)
    - “As fast as possible” is not a specification (except for libraries)
- What?
  - 90/10 rule: 10% of a program use 90% of the resources
  - Optimize these 10% only!
- How?
  - Optimizations can be
    - Algorithm oriented (e. g. O(log n) instead of O(n))
    - Hardware oriented (e. g. loop optimizations)
  - Measure, think, optimize, measure again, document
    - Do not start “optimized”

Cache

- Cache lines
  - Granularity and alignment of memory transfers
  - Use all data within a cache line
    - Data alignment
    - Data structure reorganization, e. g. arrays of structs vs. struct of arrays
    - Modify data structure traversal
- Cache associativity
  - Memory cannot be stored at arbitrary locations within the cache
    - Commonly used: least significant bits determine position in the cache
    - N-way set associative -> position can hold N cache lines
  - Cache Thrashing: accessing only memory locations mapped to one set
    - Avoid strides that are a multiple of the cache (bank) size

Loop optimizations

- Many different techniques
  - Loop unrolling
  - Loop fusion / loop distribution
  - Loop interchange
  - Loop blocking / loop unroll and fuse
  - …
- Can only be applied if data dependencies allow it
- Goals
  - Optimize data traversal (cache/prefetching hardware)
  - More (independent) instructions to reorder

Loop unroll and fuse/jam

> Original code

```
for( i=0; i<n; ++i )
    for( j=0; j<n; ++j )
        a[i][j] = b[j][i];
```

> i-loop unrolled twice

```
for( i=0; i<n-2+1; i+=2 ) {
    for( j=0; j<n; ++j )
        a[i+0][j] = b[j][i+0];
    for( j=0; j<n; ++j )
        a[i+1][j] = b[j][i+1];
}
for( ; i<n; ++i )
    for( j=0; j<n; ++j )
        a[i][i] = b[j][i];
```

> j-loops fused

```
for( i=0; i<n-2+1; i+=2 ) {
    for( j=0; j<n; ++j ) {
        a[i+0][j] = b[j][i+0];
        a[i+1][j] = b[j][i+1];
    }
}
for( ; i<n; ++i )
    for( j=0; j<n; ++j )
        a[i][j] = b[j][i];
```

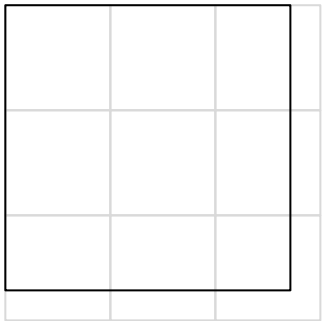
Loop tiling/blocking

> Original code

```
for( i=0; i<n; ++i )
    for( j=0; j<n; ++j )
        a[i][j] = b[j][i];
```

> Blocked loops

```
for( bi=0; bi<n; bi+=bsi )
    for( bj=0; bj<n; bj+=bsj )
        for( i=bi; i<min(n,bi+bsi); ++i )
            for( j=bj; j<min(n,bj+bsj); ++j )
                a[i][j] = b[j][i];
```



> Formulated differently

```
for( bi=0; bi<n-bsi+1; bi+=bsi ) {
    for( bj=0; bj<n-bsj+1; bj+=bsj )
        for( i=bi; i<bi+bsi; ++i )
            for( j=bj; j<bj+bsj; ++j )
                a[i][j] = b[j][i];
    for( i=bi; i<bi+bsi; ++i )
        for( j=bj; j<n; ++j )
            a[i][j] = b[j][i];
}
for( bj=0; bj<n-bsj+1; bj+=bsj )
    for( i=bi; i<n; ++i )
        for( j=bj; j<bj+bsj; ++j )
            a[i][j] = b[j][i];
for( i=bi; i<n; ++i )
    for( j=bj; j<n; ++j )
        a[i][j] = b[j][i];
```

From tiling to unroll and fuse

1. Tiled loop
   (n dividable by bsj and bsj)

```
for( bi=0; bi<n; bi+=bsi )
   for( bj=0; bj<n; bj+=bsj )
      for( i=bi; i<bi+bsi; ++i )
         for( j=bj; j<bj+bsj; ++j )
            a[i][j] = b[j][i];
```

2. Set bsj to 1, bsi to 2

```
for( bi=0; bi<n; bi+=2 )
   for( bj=0; bj<n; ++bj )
      for( i=bi; i<bi+2; ++i )
         for( j=bj; j<bj+1; ++j )
            a[i][j] = b[j][i];
```

3. Remove unnecessary loop

```
for( bi=0; bi<n; bi+=2 )
   for( bj=0; bj<n; ++bj )
      for( i=bi; i<bi+2; ++i)
         a[i][bj] = b[bj][i];
```

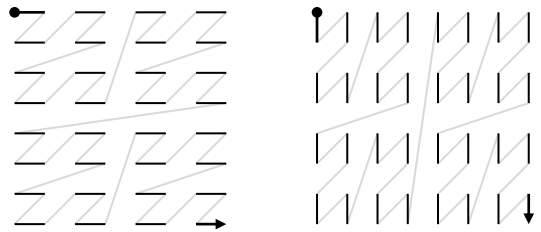
4. Completely unroll inner loop

```
for( bi=0; bi<n; bi+=2 ) {
   for( bj=0; bj<n; ++bj ) {
      a[bi+0][bj] = b[bj][bi+0];
      a[bi+1][bj] = b[bj][bi+1];
   }
}
```

Cache oblivious algorithms

- Work well on a wide range of cache hierarchies
  - Achieved by successively processing smaller data sets

```
transpose( il, ir, jl, jr ) {
    if( ir-il < threshold ) {
        transpose_block( il, ir, jl, jr );
        return;
    }
    im = il + (ir-il)/2;
    jm = jl + (jr-jl)/2;
    transpose( il, im, jl, jm );
    transpose( im, ir, jl, jm );
    transpose( il, im, jm, jr );
    transpose( im, ir, jm, jr );
}
```



Pointer aliasing

- Referencing the same memory with different pointers
  - The possibility hinders the compiler to optimize

```
void f(int *a, int *b) {
        for( i=0; i<N; ++i )
                a[i] = b[i]*3;
}
```

    - Could be called as f(&x[1], &x[0])

  - Tell the compiler that there is no aliasing

```
void f(int * restrict a, int * restrict b) {
        for( i=0; i<N; ++i )
                a[i] = b[i]*3;
}
```

    - You guarantee that all restricted pointers do not alias each other

Using SIMD Instructions

- Include file *x86intrin.h* for (activated) SSE intrinsics
  - Intrinsics == processor instructions as functions
  - Operate on vector datatypes
    - E. g. datatype for two (adjacent) doubles: __m128d
- GCC has built-in support for pair-wise operations
  - __m128d a = b + c;
- Most intrinsics require variables to be 16 byte aligned
  - __m128d *a = _mm_malloc(N * sizeof(double), 16);
  - (be careful with 2D arrays with an odd number of colums)

Summary

- ILP necessary to achieve high performance
- Compiler does most micromanagement
  - (and some macromanagement)
  - If instructed properly
- You have to do most macromanagement
  - Provide even more independent instructions
  - Optimize data structures and traversals