Optimal utilization of parallel resources

Exploiting SIMD parallelism and reading assembly code
Programming for ccNUMA memory architecture
Simultaneous multi-threading (SMT): facts & myths
Coding for Single Instruction Multiple Data processing
SIMD processing – Basics

- Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on “wide” registers.
- x86 SIMD instruction sets:
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
- Adding two registers holding double precision floating point operands

(c) RRZE 2013
SIMD processing – Basics

- **Steps (done by the compiler) for “SIMD processing”**

```java
for(int i=0; i<n; i++)
    C[i] = A[i] + B[i];
```

“Loop unrolling”

```java
for(int i=0; i<n; i+=4){
    C[i] = A[i] + B[i];
    C[i+1] = A[i+1] + B[i+1];
}
```

//remainder loop handling

Load 256 Bits starting from address of A[i] to register R0

Add the corresponding 64 Bit entries in R0 and R1 and store the 4 results to R2

Store R2 (256 Bit) to address starting at C[i]

LABEL1:

VLOAD R0 ← A[i]
VLOAD R1 ← B[i]
V64ADD[R0, R1] → R2
VSTORE R2 → C[i]
i ← i+4
i < (n-4)? JMP LABEL1
//remainder loop handling
No SIMD vectorization for loops with data dependencies:

```c
for(int i=0; i<n;i++)
    A[i]=A[i-1]*s;
```

“Pointer aliasing” may prevent SIMDfication

```c
void scale_shift(double *A, double *B, double *C, int n) {
    for(int i=0; i<n; ++i)
        C[i] = A[i] + B[i];
}
```

- C/C++ allows that \( A \rightarrow \&C[-1] \) and \( B \rightarrow \&C[-2] \)
  \( \rightarrow C[i] = C[i-1] + C[i-2] \): dependency \( \rightarrow \) No SIMD

If “pointer aliasing” is not used, tell it to the compiler:
- `-fno-alias` (Intel), `-Msafeptr` (PGI), `-fargument-noalias` (gcc)
- `restrict` keyword (C only!):
  `void f(double restrict *a, double restrict *b) {...}`
Reading x86 assembly code and exploiting SIMD parallelism

Understanding SIMD execution by inspecting assembly code
SIMD vectorization how-to
Intel compiler options and features for SIMD
Sparse MVM part 3: SIMD-friendly data layouts
Why check the assembly code?

- Sometimes the only way to make sure the compiler “did the right thing”
  - Example: “LOOP WAS VECTORIZED” message is printed, but Loads & Stores may still be scalar!

- Get the assembler code (Intel compiler):
  
  ```
  icc -S -O3 -xHost triad.c -o a.out
  ```

- Disassemble Executable:
  
  ```
  objdump -d ./a.out | less
  ```

The x86 ISA is documented in:

- Intel Software Development Manual (SDM) 2A and 2B
Basics of the x86-64 ISA

16 general Purpose Registers (64bit):
rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15
alias with eight 32 bit register set:
eax, ebx, ecx, edx, esi, edi, esp, ebp

Floating Point SIMD Registers:
xmm0-xmm15  SSE (128bit)  alias with 256-bit registers
ymm0-ymm15  AVX (256bit)

SIMD instructions are distinguished by:
AVX (VEX) prefix:   v
Operation:            mul, add, mov
Modifier:             nontemporal (nt), unaligned (u), aligned (a), high (h)
Width:                scalar (s), packed (p)
Data type:            single (s), double (d)
Case Study: Simplest code for the summation of the elements of a vector (single precision)

```c
float sum = 0.0;

for (int j=0; j<size; j++){
    sum += data[j];
}
```

To get object code use `objdump -d` on object file or executable or compile with `-S`

AT&T syntax:
```
addss 0(%rdx,%rax,4),%xmm0
```

Instruction code:
```
401d08:   f3 0f 58 04 82  addss  xmm0,[rdx + rax * 4]
401d0d:   48 83 c0 01   add    rax,1
401d11:   39 c7          cmp    edi,eax
401d13:   77 f3          ja     401d08
```

(final sum across xmm0 omitted)
Summation code (single precision): Improvements

1:
addss xmm0, [rsi + rax * 4]
add rax, 1
cmp eax, edi
js 1b

3 cycles add pipeline latency

Unrolling with sub-sums to break up register dependency

1:
addss xmm0, [rsi + rax * 4]
addss xmm1, [rsi + rax * 4 + 4]
addss xmm2, [rsi + rax * 4 + 8]
addss xmm3, [rsi + rax * 4 + 12]
add rax, 4
cmp eax, edi
js 1b

1:
vaddps ymm0,...,[rsi + rax * 4]
vaddps ymm1,...,[rsi + rax * 4 + 32]
vaddps ymm2,...,[rsi + rax * 4 + 64]
vaddps ymm3,...,[rsi + rax * 4 + 96]
add rax, 32
cmp eax, edi
js 1b

AVX SIMD vectorization

(c) RRZE 2013
How to leverage SIMD

Alternatives:
- The compiler does it for you (but: aliasing, alignment, language)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available:
- xmmintrin.h (SSE)
- pmmintrin.h (SSE2)
- immintrin.h (AVX)
- x86intrin.h (all instruction set extensions)
- See next slide for an example
Example: array summation using C intrinsics (SSE, single precision)

```
__m128 sum0, sum1, sum2, sum3;
__m128 t0, t1, t2, t3;
float scalar_sum;
sum0 = __mm_setzero_ps();
sum1 = __mm_setzero_ps();
sum2 = __mm_setzero_ps();
sum3 = __mm_setzero_ps();

for (int j=0; j<size; j+=16){
    t0 = __mm_loadu_ps(data+j);
    t1 = __mm_loadu_ps(data+j+4);
    t2 = __mm_loadu_ps(data+j+8);
    t3 = __mm_loadu_ps(data+j+12);
    sum0 = __mm_add_ps(sum0, t0);
    sum1 = __mm_add_ps(sum1, t1);
    sum2 = __mm_add_ps(sum2, t2);
    sum3 = __mm_add_ps(sum3, t3);
}
sum0 = __mm_add_ps(sum0, sum1);
sum0 = __mm_add_ps(sum0, sum2);
sum0 = __mm_add_ps(sum0, sum3);
sum0 = __mm_hadd_ps(sum0, sum0);
sum0 = __mm_hadd_ps(sum0, sum0);
_mm_store_ss(&scalar_sum, sum0);
```

summation of partial results

core loop (bulk)
Example: array summation from intrinsics, instruction code

```
14: 0f 57 c9          xorps %xmm1,%xmm1
17: 31 c0            xor %eax,%eax
19: 0f 28 d1          movaps %xmm1,%xmm2
1c: 0f 28 c1          movaps %xmm1,%xmm0
1f: 0f 28 d9          movaps %xmm1,%xmm3
22: 66 0f 1f 44 00 00  nopw 0x0(%rax,%rax,1)
28: 0f 10 3e          movups (%rsi),%xmm7
2b: 0f 10 76 10      movups 0x10(%rsi),%xmm6
2f: 0f 10 6e 20      movups 0x20(%rsi),%xmm5
33: 0f 10 66 30      movups 0x30(%rsi),%xmm4
37: 83 c0 10         add $0x10,%eax
3a: 48 83 c6 40       add $0x40,%rsi
3e: 0f 58 df         addps %xmm7,%xmm3
41: 0f 58 c6         addps %xmm6,%xmm0
44: 0f 58 d5         addps %xmm5,%xmm2
47: 0f 58 cc         addps %xmm4,%xmm1
4a: 39 c7            cmp %eax,%edi
4c: 77 da            ja 28 <compute_sum_SSE+0x18>
4e: 0f 58 c3         addps %xmm3,%xmm0
51: 0f 58 c2         addps %xmm2,%xmm0
54: 0f 58 c1         addps %xmm1,%xmm0
57: f2 0f 7c c0      haddps %xmm0,%xmm0
5b: f2 0f 7c c0      haddps %xmm0,%xmm0
5f: c3               retq
```

Loop body
Parallel resources

Vectorization and the Intel compiler

- Intel compiler will try to use SIMD instructions when enabled to do so
  - “Poor man’s vector computing”
  - Compiler can emit messages about vectorized loops (not by default):
    
    ```
    plain.c(11): (col. 9) remark: LOOP WAS VECTORIZED.
    ```

- Use option `-vec_report3` to get full compiler output about which loops were vectorized and which were not and why (data dependencies!)
- Some obstructions will prevent the compiler from applying vectorization even if it is possible

- You can use source code directives to provide more information to the compiler
Vectorization compiler options

- The compiler will vectorize starting with –O2.
- To enable specific SIMD extensions use the –x option:
  - –xSSE2 vectorize for SSE2 capable machines
    Available SIMD extensions:
    SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX
  
- –xAVX on Sandy Bridge processors

Recommended option:
- –xHost will optimize for the architecture you compile on

On AMD Opteron: use plain –O3 as the –x options may involve CPU type checks.
Vectorization compiler options

- **Controlling non-temporal stores (part of the SIMD extensions)**

  - `-opt-streaming-stores` *always|auto|never*

    - **always** use NT stores, assume application is memory bound (use with caution!)
    - **auto** compiler decides when to use NT stores
    - **never** do not use NT stores unless activated by source code directive
Rules for vectorizable loops

1. Countable
2. Single entry and single exit
3. Straight line code
4. No function calls (exception intrinsic math functions)

Better performance with:
1. Simple inner loops with unit stride
2. Minimize indirect addressing
3. Align data structures (SSE 16 bytes, AVX 32 bytes)
4. In C use the restrict keyword for pointers to rule out aliasing

Obstacles for vectorization:
- Non-contiguos memory access
- Data dependencies
User mandated vectorization

- Since Intel Compiler 12.0 the `simd` pragma is available
- `#pragma simd` enforces vectorization where the other pragmas fail
- Prerequisites:
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- There are additional clauses: reduction, vectorlength, private
- Refer to the compiler manual for further details

```c
#pragma simd reduction(+:x)
    for (int i=0; i<n; i++) {
        x = x + A[i];
    }
```

- NOTE: Using the `#pragma simd` the compiler may generate incorrect code if the loop violates the vectorization rules!
**x86 Architecture:**

**SIMD and Alignment**

- **Alignment issues**
  - Alignment of arrays with SSE (AVX) should be on 16-byte (32-byte) boundaries to allow packed aligned loads and NT stores *(for Intel processors)*
    - AMD has a scalar nontemporal store instruction
  - Otherwise the compiler will revert to unaligned loads and not use NT stores – even if you say vector nontemporal
  - Modern x86 CPUs have less (not zero) impact for misaligned LD/ST, but Xeon Phi relies heavily on it!
  - How is manual alignment accomplished?

- **Dynamic allocation of aligned memory** *(align = alignment boundary)*:

```c
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr, 
                   size_t align, 
                   size_t size);
```
Efficient parallel programming on ccNUMA nodes

Performance characteristics of ccNUMA nodes
First touch placement policy
C++ issues
ccNUMA locality and dynamic scheduling
ccNUMA locality beyond first touch


ccNUMA performance problems
“The other affinity” to care about

- ccNUMA:
  - Whole memory is transparently accessible by all processors
  - but physically distributed
  - with varying bandwidth and latency
  - and potential contention (shared memory paths)

- How do we make sure that memory access is always as "local" and "distributed" as possible?

- Page placement is implemented in units of OS pages (often 4kB, possibly more)
Cray XE6 Interlagos node
4 chips, two sockets, 8 threads per ccNUMA domain

- **ccNUMA map**: Bandwidth penalties for remote access
  - Run 8 threads per ccNUMA domain (1 chip)
  - Place memory in different domain → 4x4 combinations
  - STREAM triad benchmark using nontemporal stores

![STREAM triad performance graph]

(c) RRZE 2013
numactl as a simple ccNUMA locality tool:
How do we enforce some locality of access?

- **numactl** can influence the way a binary maps its memory pages:

  ```
  numactl --membind=<nodes> a.out  # map pages only on <nodes>
  --preferred=<node> a.out        # map pages on <node>
  # and others if <node> is full
  --interleave=<nodes> a.out      # map pages round robin across
  # all <nodes>
  ```

- **Examples:**

  ```bash
  for m in `seq 0 3`; do
      for c in `seq 0 3`; do
          env OMP_NUM_THREADS=8 \ 
              numactl --membind=$m --cpunodebind=$c ./stream
      done
  done

  env OMP_NUM_THREADS=4 numactl --interleave=0-3 \ 
      likwid-pin -c N:0,4,8,12 ./stream
  ```

- **But what is the default without** **numactl**?
ccNUMA default memory locality

- "Golden Rule" of ccNUMA:

  A memory page gets mapped into the local memory of the processor that first touches it!

  - Except if there is not enough local memory available
  - This might be a problem, see later

- Caveat: "touch" means "write", not "allocate"

- Example:

  ```c
  double *huge = (double*)malloc(N*sizeof(double));
  for(i=0; i<N; i++) // or i+=PAGE_SIZE
    huge[i] = 0.0;
  ```

- It is sufficient to touch a single item to map the entire page
Coding for ccNUMA data locality

- Most simple case: explicit initialization

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

A = 0.d0

 !$OMP parallel do
 do i = 1, N
   B(i) = function ( A(i) )
 end do
 !$OMP end parallel do
...

integer, parameter :: N = 10000000
double precision A(N), B(N)

 !$OMP parallel
 !$OMP do schedule(static)
 do i = 1, N
   A(i) = 0.d0
 end do
 !$OMP end do
 !$OMP end parallel
```
Coding for ccNUMA data locality

- Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O

```fortran
integer, parameter :: N = 10000000
double precision A(N), B(N)

READ(1000) A

!$OMP parallel do
do i = 1, N
    B(i) = function ( A(i) )
end do
!$OMP end parallel do

integer, parameter :: N = 10000000
double precision A(N), B(N)

!$OMP parallel
!$OMP do schedule(static)
do i = 1, N
    A(i) = 0.d0
end do
!$OMP end do
!$OMP single
READ(1000) A
!$OMP end single
!$OMP do schedule(static)
do i = 1, N
    B(i) = function ( A(i) )
end do
!$OMP end do
!$OMP end parallel
```

 Sometimes initialization is not so obvious: I/O cannot be easily parallelized, so “localize” arrays before I/O.
Coding for Data Locality

- **Required condition**: OpenMP loop schedule of initialization must be the same as in all computational loops
  - Only choice: `static`! Specify explicitly on all NUMA-sensitive loops, just to be sure...
  - Imposes some constraints on possible optimizations (e.g. load balancing)
  - Presupposes that all worksharing loops with the same loop length have the same thread-chunk mapping
  - If dynamic scheduling/tasking is unavoidable, more advanced methods may be in order
    - See below
- **How about global objects?**
  - Better not use them
  - If communication vs. computation is favorable, might consider properly placed copies of global data
- **C++: Arrays of objects and std::vector<> are by default initialized sequentially**
  - STL allocators provide an elegant solution
Don't forget that constructors tend to touch the data members of an object. Example:

```cpp
class D {
    double d;
public:
    D(double _d=0.0) throw() : d(_d) {}
    inline D operator+(const D& o) throw() {
        return D(d+o.d);
    }
    inline D operator*(const D& o) throw() {
        return D(d*o.d);
    }
...
};

→ placement problem with
D* array = new D[1000000];
```
Coding for Data Locality:
Parallel first touch for arrays of objects

- **Solution**: Provide overloaded `D::operator new[]`

```cpp
void* D::operator new[](size_t n) {
    char *p = new char[n];    // allocate
    size_t i,j;
    #pragma omp parallel for private(j) schedule(....)
    for(i=0; i<n; i += sizeof(D))
        for(j=0; j<sizeof(D); ++j)
            p[i+j] = 0;
    return p;
}

void D::operator delete[](void* p) throw() { 
    delete [] static_cast<char*>(p);
}
```

- **Placement of objects is then done automatically by the C++ runtime via “placement new”**
Coding for Data Locality:  
NUMA allocator for parallel first touch in std::vector<>
Diagnosing Bad Locality

- If your code is cache-bound, you might not notice any locality problems

- Otherwise, bad locality limits scalability at very low CPU numbers (whenever a node boundary is crossed)
  - If the code makes good use of the memory interface
  - But there may also be a general problem in your code…

- Running with `numactl --interleave` might give you a hint
  - See later

- Consider using performance counters
  - LIKWID-perfctr can be used to measure nonlocal memory accesses
  - Example for Intel Westmere dual-socket system (Core i7, hex-core):

    ```
    env OMP_NUM_THREADS=12 likwid-perfctr -g MEM -C N:0-11 ./a.out
    ```
Using performance counters for diagnosing bad ccNUMA access locality

- **Intel Westmere EP node (2x6 cores):**

  Only one memory BW per socket ("Uncore")

<table>
<thead>
<tr>
<th>Metric</th>
<th>core 0</th>
<th>core 1</th>
<th>core 6</th>
<th>core 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime [s]</td>
<td>0.730168</td>
<td>0.733754</td>
<td>0.732808</td>
<td>0.732943</td>
</tr>
<tr>
<td>CPI</td>
<td>10.4164</td>
<td>10.2654</td>
<td>10.5002</td>
<td>10.7641</td>
</tr>
<tr>
<td>Memory bandwidth [MBytes/s]</td>
<td>11880.9</td>
<td>0</td>
<td>...</td>
<td>11732.4</td>
</tr>
<tr>
<td>Remote Read BW [MBytes/s]</td>
<td>4219</td>
<td>0</td>
<td></td>
<td>4163.45</td>
</tr>
<tr>
<td>Remote Write BW [MBytes/s]</td>
<td>1706.19</td>
<td>0</td>
<td></td>
<td>1705.09</td>
</tr>
<tr>
<td>Remote BW [MBytes/s]</td>
<td>5925.19</td>
<td>0</td>
<td></td>
<td>5868.54</td>
</tr>
</tbody>
</table>

Half of BW comes from other socket!
If all fails...

- Even if all placement rules have been carefully observed, you may still see nonlocal memory traffic. Reasons?

  - Program has erratic access patterns → may still achieve some access parallelism (see later)
  - OS has filled memory with buffer cache data:

```
# numactl --hardware    # idle node!
available: 2 nodes (0-1)
node 0 size: 2047 MB
node 0 free: 906 MB
node 1 size: 1935 MB
node 1 free: 1798 MB
```

```
top - 14:18:25 up 92 days,  6:07,  2 users,  load average: 0.00, 0.02, 0.00
Mem:  4065564k total, 1149400k used,  2716164k free,  43388k buffers
Swap: 2104504k total,  2656k used,  2101848k free, 1038412k cached
```
ccNUMA problems beyond first touch:  
*Buffer cache*

- **OS uses part of main memory for disk buffer (FS) cache**
  - If FS cache fills part of memory, apps will probably allocate from foreign domains
  - → non-local access!
  - “sync” is not sufficient to drop buffer cache blocks

- **Remedies**
  - Drop FS cache pages after user job has run (admin’s job)
    - seems to be automatic after aprun has finished on Crays
  - User can run “sweeper” code that allocates and touches all physical memory before starting the real application
  - *numactl* tool or *aprun* can force local allocation (where applicable)
  - Linux: There is no way to limit the buffer cache size in standard kernels
ccNUMA problems beyond first touch:
Buffer cache

Real-world example: ccNUMA and the Linux buffer cache

Benchmark:
1. Write a file of some size from LD0 to disk
2. Perform bandwidth benchmark using all cores in LD0 and maximum memory installed in LD0

Result: By default, Buffer cache is given priority over local page placement → restrict to local domain if possible!
The curse and blessing of interleaved placement:  
*OpenMP STREAM on a Cray XE6 Interlagos node*

- **Parallel init**: Correct parallel initialization
- **LD0**: Force data into LD0 via `numactl --m 0`
- **Interleaved**: `numactl --interleave <LD range>`

![Graph showing performance comparison]

(c) RRZE 2013  
Parallel resources
The curse and blessing of interleaved placement:
OpenMP STREAM triad on 4-socket (48 core) Magny Cours node

- **Parallel init**: Correct parallel initialization
- **LD0**: Force data into LD0 via `numactl -m 0`
- **Interleaved**: `numactl --interleave <LD range>`

![Bar chart showing bandwidth (Mbyte/s) vs. number of NUMA domains (6 threads per domain)](chart.png)
Summary on ccNUMA issues

- **Identify the problem**
  - Is ccNUMA an issue in your code?
  - Simple test: run with `numactl --interleave`

- **Apply first-touch placement**
  - Look at initialization loops
  - Consider loop lengths and static scheduling
  - C++ and global/static objects may require special care

- **If dynamic scheduling cannot be avoided**
  - Consider round-robin placement

- **Buffer cache may impact proper placement**
  - Kick your admins
  - or apply sweeper code
  - If available, use runtime options to force local placement
Simultaneous multithreading (SMT)

Principles and performance impact
SMT vs. independent instruction streams
Facts and fiction
SMT Makes a single physical core appear as two or more “logical” cores → multiple threads/processes run concurrently

- **SMT principle (2-way example):**

  ![Diagram showing standard core and 2-way SMT](image)
SMT impact

- **SMT is primarily suited for increasing processor throughput**
  - With multiple threads/processes running concurrently
- **Scientific codes tend to utilize chip resources quite well**
  - Standard optimizations (loop fusion, blocking, …)
  - High data and instruction-level parallelism
  - Exceptions do exist

- **SMT is an important topology issue**
  - SMT threads share almost all core resources
    - Pipelines, caches, data paths
  - Affinity matters!
  - If SMT is not needed
    - pin threads to physical cores
    - or switch it off via BIOS etc.
SMT impact

- SMT adds another layer of topology (inside the physical core)
- Caveat: SMT threads share all caches!
- Possible benefit: Better pipeline throughput
  - Filling otherwise unused pipelines
  - Filling pipeline bubbles with other thread’s executing instructions:

```
Thread 0:
do i=1,N
   a(i) = a(i-1)*c
endo
```
```
Thread 1:
do i=1,N
   b(i) = s*b(i-2)+d
endo
```

- Beware: Executing it all in a single thread (if possible) may reach the same goal without SMT:

```
do i=1,N
   a(i) = a(i-1)*c
   b(i) = s*b(i-2)+d
endo
```
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
MULT Pipeline depth: 5 stages → 1 F / 5 cycles for recursive update

Fill bubbles via:
- SMT
- Multiple streams

Thread 0:
\[
\begin{align*}
&\text{do } i=1,N \\
&A(i)=A(i-1)\times c \\
&B(i)=B(i-1)\times d
\end{align*}
\]

Thread 1:
\[
\begin{align*}
&\text{do } i=1,N \\
&A(i)=A(i-1)\times c \\
&B(i)=B(i-1)\times d
\end{align*}
\]
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
MULT Pipeline depth: 5 stages $\rightarrow$ 1 F / 5 cycles for recursive update

5 independent updates on a single thread do the same job!
Simultaneous recursive updates with SMT

Intel Sandy Bridge (desktop) 4-core; 3.5 GHz; SMT
Pure update benchmark can be vectorized $\rightarrow$ 2 F / cycle (store limited)

Recursive update:
- SMT can fill pipeline bubbles
- A single thread can do so as well
- Bandwidth does not increase through SMT
- SMT can not replace SIMD!

(c) RRZE 2013
Parallel resources
SMT myths: Facts and fiction (1)

- Myth: “If the code is compute-bound, then the functional units should be saturated and SMT should show no improvement.”

- Truth
  1. A compute-bound loop does not necessarily saturate the pipelines; dependencies can cause a lot of bubbles, which may be filled by SMT threads.
  2. If a pipeline is already full, SMT will not improve its utilization.
Myth: “If the code is memory-bound, SMT should help because it can fill the bubbles left by waiting for data from memory.”

Truth:
1. If the maximum memory bandwidth is already reached, SMT will not help since the relevant resource (bandwidth) is exhausted.

2. If the relevant bottleneck is not exhausted, SMT may help since it can fill bubbles in the LOAD pipeline.

This applies also to other “relevant bottlenecks!”
Myth: “SMT can help bridge the latency to memory (more outstanding references).”

Truth:
Outstanding references may or may not be bound to SMT threads; they may be a resource of the memory interface and shared by all threads. The benefit of SMT with memory-bound code is usually due to better utilization of the pipelines so that less time gets “wasted” in the cache hierarchy.

See also the “ECM Performance Model” later on.
Things to remember

Goals for optimization:

1. **Map your work to an instruction mix with highest throughput using the most effective instructions.**

2. **Reduce data volume over slow data paths fully utilizing available bandwidth.**

3. **Avoid possible hazards/overhead which prevent reaching goals one and two.**