Algorithms in Scientific Computing II
Fundamentals –
Parallel Architectures, Models, and Languages

Michael Bader
TUM – SCCS

Winter 2011/2012
Part I

Parallel Architectures
Multicore CPUs – Intel’s Nehalem Architecture

- example: quad-core CPU with shared and private caches
- simultaneous multithreading: 8 threads on 4 cores
- memory architecture: Quick Path Interconnect (replaced Front Side Bus)
Multicore CPUs – Intel’s Nehalem Architecture (2)

- NUMA (non-uniform memory access) architecture: CPUs have “private” memory, but uniform access to remote memory
- max. 25 GB/s bandwidth

Intel® QuickPath Technology

(source: Intel – Nehalem Whitepaper)
Manycores CPU – Intel Knights Ferry

Intel® MIC Architecture:
An Intel Co-Processor Architecture

Many cores and many, many more threads
Standard IA programming and memory model

(source: Intel/K. Skaugen – SC’10 keynote presentation)
Knights Ferry

- Software development platform
- Growing availability through 2010
- 32 cores, 1.2 GHz
- 128 threads at 4 threads / core
- 8MB shared coherent cache
- 1-2GB GDDR5
- Bundled with Intel HPC tools

Software development platform for Intel® MIC architecture

(source: Intel/K. Skaugen – SC’10 keynote presentation)
GPGPU – NVIDIA Fermi

CUDA's hierarchy of threads maps to a hierarchy of processors on the GPU; a GPU executes one or more kernel grids; a streaming multiprocessor (SM) executes one or more thread blocks; and CUDA cores and other execution units in the SM execute threads. The SM executes threads in groups of 32 threads called a warp. While programmers can generally ignore warp execution for functional correctness and think of programming one thread, they can greatly improve performance by having threads in a warp execute the same code path and access memory in nearby addresses.

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

(source: NVIDIA – Fermi Whitepaper)
Third Generation Streaming Multiprocessor

The third generation SM introduces several architectural innovations that make it not only the most powerful SM yet built, but also the most programmable and efficient.

- Each SM features 32 CUDA cores, offering a fourfold increase over prior SM processors.
- The Fermi Streaming Multiprocessor (SM) design is optimized for sixteen threads per clock.
- Supporting units load and store the data at each address to minimize latency.
- Each SM has 16 load/store units, allowing source and destination addresses to be calculated simultaneously.
- Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count operations are supported.
- 64-bit and extended precision operations are available.
- Various instructions, including those necessary for high-level programming language requirements, are optimized.
- The integer ALU is also optimized to efficiently support high-level language requirements.
- The designed integer ALU supports full 32-bit precision for all instructions, consistent with standard 32-bit architectures.
- Multi-instruction emulation sequences were required for integer arithmetic in previous generations.
- In Fermi, the newly implemented integer ALU provides full 32-bit precision for all instructions.
- GT200 limited integer arithmetic precision to 24-bit for multiply operations, whereas Fermi supports full 32-bit precision.
- Fused multiply-add (FMA) is more accurate than performing the operations separately. GT200 implemented double precision FMA.
- FMA reduces the loss of precision in the addition.
- FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision.
- FMA implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision floating point arithmetic.
- Prior GPUs used IEEE 754-1985 floating-point logic unit (ALU) and floating point unit (FPU).
- In Fermi, the integer ALU is optimized to efficiently support high-level language requirements.

(source: NVIDIA – Fermi Whitepaper)
General Purpose Graphics Processing Unit:

- 512 CUDA cores
- improved double precision performance
- shared vs. global memory
- new: L1 and L2 cache (768 KB)
- trend from GPU towards CPU?

Memory Subsystem Innovations
NVIDIA Parallel DataCache™ with Configurable L1 and Unified L2 Cache

Working with hundreds of GPU computing applications from various industries, we learned that while Shared memory benefits many problems, it is not appropriate for all problems. Some algorithms map naturally to Shared memory, others require a cache, while others require a combination of both. The optimal memory hierarchy should offer the benefits of both Shared memory and cache, and allow the programmer a choice over its partitioning. The Fermi memory hierarchy adapts to both types of program behavior.

Adding a true cache hierarchy for load/store operations presented significant challenges. Traditional GPU architectures support a read-only “load” path for texture operations and a write-only “export” path for pixel data output. However, this approach is poorly suited to executing general purpose C or C++ thread programs that expect reads and writes to be ordered. As one example: spilling a register operand to memory and then reading it back creates a read after write hazard; if the read and write paths are separate, it may be necessary to explicitly flush the entire write/“export” path before it is safe to issue the read, and any caches on the read path would not be coherent with respect to the write data.

The Fermi architecture addresses this challenge by implementing a single unified memory request path for loads and stores, with an L1 cache per SM multiprocessor and unified L2 cache that services all operations (load, store and texture). The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations. The 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache, or 16 KB of Shared memory with 48 KB of L1 cache. When configured with 48 KB of shared memory, programs that make extensive use of shared memory (such as electrodynamic simulations) can perform up to three times faster. For programs whose memory accesses are not known beforehand, the 48 KB L1 cache configuration offers greatly improved performance over direct access to DRAM.
<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>Computer/Year Vendor</th>
<th>Cores</th>
<th>$R_{\text{max}}$</th>
<th>$R_{\text{peak}}$</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RIKEN Advanced Institute for Computational Science (AICS) Japan</td>
<td>K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu</td>
<td>548352</td>
<td>8162.00</td>
<td>8773.63</td>
<td>9898.56</td>
</tr>
<tr>
<td>2</td>
<td>National Supercomputing Center in Tianjin China</td>
<td>Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT</td>
<td>186368</td>
<td>2566.00</td>
<td>4701.00</td>
<td>4040.00</td>
</tr>
<tr>
<td>3</td>
<td>DOE/SC/Oak Ridge National Laboratory United States</td>
<td>Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.</td>
<td>224162</td>
<td>1759.00</td>
<td>2331.00</td>
<td>6950.60</td>
</tr>
<tr>
<td>4</td>
<td>National Supercomputing Centre in Shenzhen (NSCS) China</td>
<td>Nebulae - Dawning TC3600 Blade, Intel X5650, Nvidia Tesla C2050 GPU / 2010 Dawning</td>
<td>120640</td>
<td>1271.00</td>
<td>2984.30</td>
<td>2580.00</td>
</tr>
<tr>
<td>5</td>
<td>GSIC Center, Tokyo Institute of Technology Japan</td>
<td>TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP</td>
<td>73278</td>
<td>1192.00</td>
<td>2287.63</td>
<td>1398.61</td>
</tr>
<tr>
<td>6</td>
<td>DOE/NNSA/LANL/SNL United States</td>
<td>Cielo - Cray XE6 8-core 2.4 GHz / 2011 Cray Inc.</td>
<td>142272</td>
<td>1110.00</td>
<td>1365.81</td>
<td>3980.00</td>
</tr>
<tr>
<td>7</td>
<td>NASA/Ames Research Center/NAS United States</td>
<td>Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband / 2011 SGI</td>
<td>111104</td>
<td>1088.00</td>
<td>1315.33</td>
<td>4102.00</td>
</tr>
<tr>
<td>8</td>
<td>DOE/SC/LBNL/NERSC United States</td>
<td>Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.</td>
<td>153408</td>
<td>1054.00</td>
<td>1288.63</td>
<td>2910.00</td>
</tr>
<tr>
<td>9</td>
<td>Commissariat a l'Energie Atomique (CEA) France</td>
<td>Tera-100 - Bull bulx super-node S6010/S6030 / 2010 Bull SA</td>
<td>138368</td>
<td>1050.00</td>
<td>1254.55</td>
<td>4590.00</td>
</tr>
<tr>
<td>10</td>
<td>DOE/NNSA/LANL United States</td>
<td>Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM</td>
<td>122400</td>
<td>1042.00</td>
<td>1375.78</td>
<td>2345.50</td>
</tr>
</tbody>
</table>
Roadrunner (Los Alamos National Lab.)

- #1 of the Top 500, years 2008–2009
- costs of installation: ~100 Mio$
- first PetaFlop supercomputer; hybrid architecture:
  - 6,562 dual-core Opteron
  - 12,240 Cell Broadband Engine
- 1.026 PFlop/s (Linpack benchmark); 98 TB memory
- power consumption: 2.35 MW ⇒ 437 MFlop/s per Watt(!)
- applications (among others): “ensure safety and reliability of nation’s nuclear weapons stockpile”

further info (architecture/programming/applications):
K computer (RIKEN, Japan)

- current #1 of the Top 500
- built by Fujitsu
- \( \approx 8 \) PFlop/s (Linpack benchmark; 93% of peak!); extension to 10 PFlop/s planned
  \( \Rightarrow \) largest performance jump in the last years
- \( \approx 550,000 \) cores (SPARC64; 2.0 GHz);
  custom architecture: 8-core processors (4 procs. per node)
- water-cooled CPUs
- power consumption 9.9 MW \( \Rightarrow \) 0.8 GFlop/s per Watt
Part II

Parallel Models
The PRAM Model(s)

Concurrent or Exclusive Read/Write Access:

- **EREW** exclusive read, exclusive write
- **CREW** concurrent read, exclusive write
- **ERCW** exclusive read, concurrent write
- **CRCW** concurrent read, concurrent write
Exclusive/Concurrent Read and Write Access

exclusive read

exclusive write

concurrent read

concurrent write
Example: Minimum Search on the PRAM

“Binary Fan-In”:
Minimum on the PRAM – Implementation

MinimumPRAM( L:Array[1..n]) : Integer {
  ! n assumed to be 2^k
  ! Model: EREW PRAM
  for i from 1 to k do {
    for j from 1 to n/(2^i) do in parallel
      else L[j] := L[2j−1];
    end if;
  }
  return L[1];
}

Complexity: \( T(n) = \Theta(\log n) \) on \( \frac{n}{2} \) processors
Parallel External Memory – Memory Scheme

[Arge, Goodrich, Nelson, Sitchinava, 2008]
Parallel External Memory – History

Extension of the classical I/O model:

- large, global memory (main memory, hard disk, etc.)
- CPU can only access smaller working memory (cache, main memory, etc.) of $M$ words each
- both organised as cache lines of size $B$ words
- algorithmic complexity determined by memory transfers

Extension of the PRAM:

- multiple CPUs access global shared memory (but locally distributed)
- EREW, CREW, CRCW classification (for local and external memory)
- similar programming model (synchronized execution, e.g.)
Bulk Synchronous Parallelism

- suggested as a “bridging model” between software and hardware aspects
- hardware model:
  - multiple CPUs with private memory
  - CPUs connected via point-to-point network

![Diagram showing multiple CPUs connected via a point-to-point network]
Bulk Synchronous Parallelism – Execution Model

Computation organised into sequence of “Super Steps”:

1. each CPU executes a sequence of operations (on local data, synchronised from the last super step)
2. CPUs send and receive point-to-point messages (no broadcasts or send/receive to/by multiple CPUs allowed)
3. synchronisation at a barrier

Goal:

- estimate time for steps 1, 2, 3 based on CPU speed, bandwidth, and latency
Interconnection Networks

- multiple CPUs with private memory
- CPUs connected via interconnection network
- new: topology of the network explicitly considered

Example: 1D mesh (linear array) of processors:
Problem: Broadcast

- information transported by at most 1 processor per step
- phase 1: propagate along first line
- phase 2: propagate along columns
Broadcast 2D mesh (P[1,1]:X, n) {

! Model: 2D mesh p[i,j] with n*n processors

input: P[1,1]:X ! element to be broadcasted

for j from 1 to n−1 do
    P[1,j+1]:X <<< P[1,j]:X

for i from 1 to n−1 do
    for P[i,j]: 1<=j<=n do in parallel
        P[i+1,j]:X <<< P[i,j]:X
    end in parallel

output: P[i,j]:X ! X available on each processor

}

Time complexity: 2n − 2 steps on n processors
Part III

Parallel Languages
OpenMP

- shared-memory application programming interface (API)
- extends *sequential* programs by directives to help compiler generate parallel code
- available for Fortran or C/C++
- *fork-join-model*: programs will be executed by a team of cooperating threads
- memory is shared between threads, except few private variables
void mvp(int m, int n, double* restrict y,
        double** restrict A, double* restrict x)
{
    int i, j;

    #pragma omp parallel for default(none) \
        shared(m,n,y,A,x) private(i,j)
    for (i=0; i<n; i++) {
        y[i] = 0.0;
        for (j=0; j<n; j++) {
            y[i] += A[i][j]*x[j];
        }
    } /*--- end of omp parallel for ---*/
}
OpenMP Directives

OpenMP directives are inserted as `#pragma`:

```
#pragma omp parallel for default(none) \ 
    shared(m,n,y,A,x) private(i,j)
```

Advantages:

- directives will be ignored by compilers that do not support OpenMP
- sequential and parallel program in the same code!
- incremental programming approach possible
  (add parallel code sections as required)
OpenMP’s Memory Model

```c
#pragma omp parallel ... default(none) \ 
     shared(m,n,y,A,x) private(i,j)
```

- memory usually shared between threads – here: matrix and vectors
- however, certain variables are private: loop variables (here: indices), temporary variables, etc.
- if not specified, default settings apply – here: `default(none)` to switch off all default settings
- programmer is responsible to sort out concurrent accesses! (even if default settings are used)
Pthreads

- standardised programming interface according to POSIX (Portable Operating System Interface)
- thread model is a generalised version of the UNIX process model (forking of threads on shared address space)
- scheduling of threads to CPU cores done by operating system
Pthreads – Typical Methods

- `pthread_create(...)`: (main) thread creates a further thread that will start by executing a specified function
- `pthread_create(...)`: thread will wait until a specified thread has terminated (useful for synchronisation)
- `pthread_cancel(...)`: cancel another thread
- Functions to synchronise data structures:
  - `mutex`: mutual exclusive access to data structures;
  - `cond`: wait for or signal certain conditions
- Functions to take influence on scheduling
- Etc.
Programming Patterns

Pthreads allow arbitrary coordination of threads; however, certain programming patterns are common, e.g.:

- **Master-Slave** model:
  master thread controls program execution and parallelisation by delegating work to slave threads

- **Worker** model:
  threads are not hierarchically organised, but distribute/organise the operations between themselves (example: jobs retrieved from a work pool)

- **Pipelining** model:
  threads are organised via input/output relations: certain threads provide data for others, etc.
#include <pthread.h>

typedef struct {
    int size, row, column;
    double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void thread_mult(matrix_type_t *work) {
    int i, row = work->row, col = work->column;
    work->MC[row][col] = 0.0;
    for (i = 0; i < work->size; i++)
        work->MC[row][col] +=
            work->MA[row][i] * work->MB[i][col];
}
Example: Matrix Multiplication (cont.)

```c
void main() {
    double MA[8][8], MB[8][8], MC[8][8];
    pthread_t thread [8*8];
    for(int row=0; row<8;row++)
        for(int col=0; col<8;col++) {
            matrix_type_t *work = (matrix_type_t *) malloc( /* ... */ );
            work->size = 8; work->row = row; work->col = col;
            work->MA = MA; work->MB = MB; work->MC = MC;
            pthread_create (&(thread[col+8*row]), NULL,
                            (void*) thread_mult, (void*) work);
        }
    for(int i=0; i<8*8;i++) pthread_join(thread[i], NULL);
}
```

(example from: Rauber&Rünger: Parallele Programmierung)
Java Threads

- object-oriented language design explicitly includes threads
- class Thread to represent threads – can be extended to implement customised threads (inherits start(), run() methods, etc.)
- interface Runnable:
  classes that implement Runnable, i.e., provide a method run() can be used to create a thread:
  \[
  \text{Thread th = new Thread(runnableObject)};
  \]
- keyword synchronized for methods that should be treated as a critical region
- methods wait() and notify() in class Object
Example: Matrix Multiplication

class MatMult extends Thread {
    static int a [], b [], c [], n=3;
    int row;
    MatMult(int _row) {
        row = _row; this . start ();
    }
    public void run() {
        for (int i=0; i<n; i++) {
            c[row][i] = 0.0;
            for (int j=0; j<n; j++)
                c[row][i] = c[row][i] + a[row][j]*b[j][i];
        }
    }
} /* class MatMult t.b.c. */
Example: Matrix Multiplication (cont.)

```java
public static void main() {
    a = new int[n][n]; b = new int[n][n]; c = new int[n][n];
    /* ... initialise a,b,c ... */
    MatMult mat = new MatMult[n];
    for(int i=0; i<n; i++) mat[i] = new MatMult(i);
    try {
        for(int i=0; i<n; i++) mat[i].join();
    } catch(Exception E) { /* ... */ }
}
```

(cmp. example in Rauber&Rünger: Parallele Programmierung)
Unified Parallel C (UPC)

- extension of C, specified in the ISO C99 standard
- based on a *distributed shared memory* model: physically distributed memory with shared address space
- **PGAS** language: “partitioned global address space”
- single program, multiple data: every program is executed in parallel on specified number of threads
- variables are private by default, but can be declared as shared
- consistency model can be varied: strict vs. relaxed
Example: Matrix Multiplication

Declaration of variables:

\textbf{shared} \ [N\times N/\text{THREADS}] \ \textbf{int} \ a[N][N];
\textbf{shared} \ [N/\text{THREADS}] \ \textbf{int} \ b[N][N];
\textbf{shared} \ [N\times N/\text{THREADS}] \ \textbf{int} \ c[N][N];

Variables have an affinity towards threads:

- \textit{block-cyclic} distribution of variables top threads
- a and c declared with a block size of N\times N/\text{THREADS} → block-oriented distribution of rows to threads
- b declared with a block-size of N/\text{THREADS} → block-oriented distribution of columns to threads

Affinity can reflect physical distribution of data
Example: Matrix Multiplication (cont.)

Code excerpt for matrix multiplication:

```c
upc_forall ( i=0;i<N;i++;&a[i][0])
    /* &a[i][0] specifies that iteration will be executed by thread that has affinity to a[i][0] */
    for (j=0;j<N; j++) {
        c[i][j] = 0;
        for (l=0;l<N;l++) c[i][j] += a[i][l]*b[l][j];
    }
    upc_barrier;
```

(source: Rauber&Rünger: Parallele Programmierung)
Further PGAS Languages

- Co-Array Fortran (CAF)
  → will become part of the next Fortran standard
- Titanium (similar to UPC, but for Java)
- X10: extension of Java;
  *globally asynchronous, locally synchronous*: add “places” that execute threads
- Chapel
- Fortress
Further Example: Intel PBB

“Intel Parallel Building Blocks”:

- language extensions & libraries for C/C++
- **Intel Cilk Plus**: language extension for simple loop & task oriented parallelism for C/C++
- **Intel Threading Building Blocks**: C++ template library to support task parallelism
- **Intel Array Building Blocks**: C++ template library to support vector parallelism
Examples – Intel Cilk Plus

Intel Cilk:

```c
void mergesort(int a[], int left, int right) {
    if (left < right) {
        int mid = (left + right)/2;
        cilk_spawn mergesort(a, left, mid);
        mergesort(a, mid, right);
        cilk_sync;
        merge(a, left, mid, right);
    }
}
```

(source: Intel)
Examples – Intel ArBB

**Intel ArBB:**

```cpp
void matvec_product(const dense<f32, 2>& matrix,
                     const dense<f32>& vector,
                     dense<f32>& result)
{
    result = add_reduce(matrix
                       * repeat_row(vector, matrix.num_rows()));
}
```

(source: Intel)
Message Passing – MPI

Abstraction of a distributed memory computer

- MPI run consists of a set of processes with separate memory space
- processes can exchange data by sending messages
- no explicit view on network topology required

SIMD/MIMD?? . . . → MPMD/SPMD

- processes can run different programs (“codes”)
  → multiple program multiple data (MPMD)
- more common (and simpler):
  processes run instances of the same program (“code”)
  → single program multiple data (SPMD)
#include "mpi.h"

int main(int argc, char **argv)
{
    int myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
        send_a_message();
    else if (myrank == 1)
        receive_a_message();
    MPI_Finalize();
}
MPI Example: “Hi there” ... (cont.)

```c
void send_a_message() {
    char message[40];
    strcpy (message,"Mr. Watson, come here, I want you.");
    MPI_Send(message, strlen(message)+1, MPI_CHAR, 1, 110, MPI_COMM_WORLD);
}

void receive_a_message() {
    char message[40];
    MPI_Status status;
    MPI_Recv(message, 40, MPI_CHAR, 0, 110, MPI_COMM_WORLD, &status);
    printf(" received : \%s:\n", message);
}
```
References (Languages)