Announcements

• Next class cancelled (Nov 19th)

• Programming Assignment # 2 deadline
  – Official deadline: 5pm, Dec 4, 2009
  – No-penalty extension: 5pm, Dec 11, 2009

• Take-home final exam
  – To be given on Dec 11, 2009
  – Due by 5pm, Dec 16, 2009

• Optional project presentations
  – Doodle poll to be sent for Dec 14-16
Acknowledgments for today’s lecture

- Slides for today’s lecture were taken from “Lecture Slides and Hands On Programming Assignments for SDK 3.0”
  - Slides for Day 1
  - http://toonces.cc.gatech.edu/users/bader/SDK30/Day1-Students.zip
Cell Broadband Engine – An Introduction to QS21 and SDK for Multicore Acceleration Version 3.0

Cell Programming Workshop
Cell/Quasar Ecosystem & Solutions Enablement
Highlights (3.2 GHz)

- 241M transistors
- 235mm²
- 9 cores, 10 threads
- >200 GFlops (SP)
- >20 GFlops (DP)
- Up to 25 GB/s memory B/W
- Up to 75 GB/s I/O B/W
- >300 GB/s EIB
- Top frequency >4GHz (observed in lab)
IBM BladeCenter QS21

- **Core Electronics**
  - Dual 3.2GHz Cell/B.E. Processor Configuration
  - 2GB XDRAM (1GB per processor)
  - Dual Gigabit Ethernet (GbE) controllers
  - Single-wide blade (uses 1 BladeCenter H slot)
  - Infiniband 4x channel adapters / (optional)
    - Cisco Systems 4X InfiniBand HCA Expansion Card for BladeCenter (32R1760)
  - Serial Attached SCSI (SAS) daughter card (39Y9190) / (optional)

- **BC Chassis Configuration**
  - Standard IBM BladeCenter H
  - Max. 14 QS21 per chassis
  - 2 Gigabit Ethernet switches
  - External IB switches required for IB option
    - Cisco Systems 4X InfiniBand Switch Module for BladeCenter (32R1756)

- **Peak Performance**
  - Up to 460 GFLOPS per blade
  - Up to 6.4 TFLOPS (peak) in a single BladeCenter H chassis
  - Up to 25.8 TFLOPS in a standard 42U rack
IBM to Build World's First Cell Broadband Engine™ Based Supercomputer

Revolutionary Hybrid Supercomputer at Los Alamos National Laboratory Will Harness Cell Game Chips and AMD Opteron™ Technology

- x86 Linux Master Cluster
  - AMD Opteron™ System x3755

- Cell BE Accelerator Linux Cluster
  - 8000+ Blades

- Goal 1 PetaFlop Double Precision Floating Point Sustained
  - 1.6 PetaFlop Peak DP Floating point (3.2 SP)
  - 360 server racks that take up around 12,000 square feet—about three basketball courts.
  - Hybrid of Opteron X64 AMD processors (System x3755 servers) and Cell BE Blade Servers connected via high speed network
  - Modular Approach means the Master Cluster could be made up of Any Type System – including Power, Intel
Roadrunner At a Glance

- **Cluster of 18 Connected Units**
  - 6,912 AMD dual-core Opterons
  - 12,960 IBM Cell eDP accelerators
  - 49.8 Teraflops peak (Opteron)
  - 1.33 Petaflops peak (Cell eDP)
  - 1PF sustained Linpack

- **InfiniBand 4x DDR fabric**
  - 2-stage fat-tree; all-optical cables
  - Full bi-section BW within each CU
    - 384 GB/s (bi-directional)
  - Half bi-section BW among CUs
    - 3.45 TB/s (bi-directional)
  - Non-disruptive expansion to 24 CUs

- **80 TB aggregate memory**
  - 28 TB Opteron
  - 52 TB Cell

- **216 GB/s sustained File System I/O**
  - 216x2 10G Ethereans to Panasas

- **RHEL & Fedora Linux**

- **SDK for Multicore Acceleration**

- **xCAT Cluster Management**
  - System-wide GigEnet network

- **3.9 MW Power**
  - 0.35 GF/Watt

- **Area**
  - 296 racks
  - 5500 ft²
Cell/B.E.’s Sweet Code Is Structured and Predictable Data

Graphics & Video Applications
- Transformation between viewpoint; time vs space; 2D vs 3D
- Data parallel algorithms image and volume processing
- Computational geometry surfaces and modeling
- Image-based modeling and rendering
- Global illumination and lighting
- Ray tracing/Ray casting
- Visualization of output
- Advanced rendering
- Advanced graphics
- Image processing
- Video processing
- Rich media
- High definition

Floating Point Intensive Applications
- FFT
- Sonar
- Seismic
- Physics
- HPC codes
- Computer vision
- Scientific computing
- Stream processing
- Numeric algorithms
- Dynamics simulation
- Numerical algorithms
- Medical & bioinformatics

Ideal SPU Codes
- Structured, predictable data
- Multiple operations performed on data
- Easy parallelize and SIMDize
- Compute intense codes
- Code fits streaming model

Data Manipulation Applications
- Audio and signal processing
- Compression/decompression
- Encryption/decryption
- Secure transformation
Cell/B.E. Performance in Hybrid Environment

Three types of processors work together.

- parallel computing on Cell
  - data partitioning & work queue pipelining
  - process management & synchronization

- remote communication to/from Cell
  - data communication & synchronization
  - process management & synchronization
  - computationally-intensive offload

- MPI remains as the foundation

Source: John A. Turner, Roadrunner Applications Team: Cell and Hybrid Results to Date, (Los Alamos Unclassified Report LA-UR-07-7573), http://www.lanl.gov/orgs/hpc/roadrunner/rrperfassess.shtml
Cell BE SDK for Multicore Acceleration Version 3.0
GCC and GNU Toolchain

- **Base toolchain**
  - Based on GCC 4.1.1 extended by PPE and SPE support
  - binutils 2.18, SPE newlib 1.15.0+, GDB 6.6+

- **Support additional languages**
  - GNU Fortran for PPE and SPE
    - **No** SPE-specific extensions (e.g. intrinsics)
  - GNU Ada for PPE only
    - Will provide Ada bindings for **libspe2**

- **Compiler performance enhancements**
  - Improved auto-vectorization capabilities
    - Extract parallelism from straight-line code, outer loops
  - Other SPE code generation improvements
    - If-conversion, modulo-scheduling enhancements

- **New hardware support**
  - Code generation for SPE with enhanced double precision FP
GCC and GNU Toolchain

- Help simplify Cell/B.E. application development
  - Syntax extension to allow use of operators (+, -, ...) on vectors
  - Additional PPU VMX intrinsics
  - Simplify embedding of SPE binaries into PPE objects
  - SPE static stack-space requirement estimation
  - Extended C99/POSIX run-time library support on SPE

- Integrated PPE address-space access on SPE
  - Syntax extension to provide address-space qualified types
  - Access PPE-side symbols in SPE code
  - Integrated software-managed cache for data access

- Combined PPE/SPE debugger enhancements
  - Extended support for debugging libspe2 code
  - Improved resolution of multiply-defined symbols
GNU tool chain

- Contains the GCC compiler for the PPU and the SPU.
  - ppu-gcc, ppu-g++, ppu32-gcc, ppu32-g++, spu-gcc, spu-g++
  - For the PPU, GCC replaces the native GCC on PPC platforms and it is a cross-compiler on x86. The GCC for the PPU is preferred and the makefiles are configured to use it when building the libraries and samples.
  - For the SPU, GCC contains a separate SPE cross-compiler that supports the standards defined in the following documents:
    - C/C++ Language Extensions for Cell BE Architecture V2.4
    - SPU Application Binary Interface (ABI) Specification V1.7
    - SPU Instruction Set Architecture V1.2

- The assembler and linker are common to both the GCC and XL C/C++ compilers.
  - ppu-ld, ppu-as, spu-ld, spu-as
  - The GCC associated assembler and linker additionally support the SPU Assembly Language Specification V1.5.
GNU tool chain (Cont’d)

- The assembler and linker are common to both the GCC and XL C/C++ compilers.
- GDB support is provided for both PPU and SPU debugging, and the debugger client can be in the same process or a remote process.
- GDB also supports combined (PPU and SPU) debugging.
  - ppu-gdb, ppu-gdbserver, ppu32-gdbserver
XL C/C++/Fortran Compilers

- **IBM XL C/C++ for Multicore Acceleration for Linux, V9.0 (dual source compiler)**
  - Product quality and support
  - Performance improvements in auto-SIMD
  - Improved diagnostic capabilities for detecting SIMD opportunities (-qreport)
  - Enablement of high optimization levels (O4, O5) on the SPE
  - Automatic generation of code overlays

- **IBM XL Fortran for Multicore Acceleration for Linux, V11.1 (dual source compiler)**
  - Beta level (with GA targeted for 11/30/07)
  - Optimized Fortran code generation for PPE and SPE
  - Support for Fortran 77, 90 and 95 standards as well as many features from the Fortran 2003 standard
  - Auto-SIMD optimizations
  - Automatic generation of code overlays

- **IBM XL C/C++ Alpha Edition for Multicore Acceleration, V0.9 (single source compiler)**
  - Beta level
  - Allows programmer to use OpenMP directives to specify parallelism on PPE and SPE
    - Compiler hides complexity of DMA transfers, code partitioning, overlays, etc.. from the programmer
IBM XL C/C++ compiler

- A cross-compiler hosted on a x86 and PPC platform.
- Requires the GCC Tool chain for cross-assembling and cross-linking applications for both the PPE and SPE.
- The XL C/C++ compiler provides the following invocation commands:
  - ppuxlc, ppuxlc++
  - spuxlc, spuxlc++
IBM XL C/C++ compiler (Cont’d)

- The XL C/C++ compiler includes the following base optimization levels:
  - -O0: almost no optimization
  - -O2: strong, low-level optimization that benefits most programs
  - -O3: intense, low-level optimization analysis with basic loop optimization
  - -O4: all of -O3 and detailed loop analysis and good whole-program analysis at link time
  - -O5: all of -O4 and detailed whole-program analysis at link time.
- Auto-SIMDization is enabled at O3 -qhot or O4 and O5 by default.
Application Developer Dilemma

PPU

Main Application

Acceleration Library

Sched. Tsk. Mg. Msg. Interface

OS

Acceleration Library:
specifies methods for cooperating with computing kernels on SPU

Task Queue Load Balancer

Task Management:
Tasks generate, management

Scheduling:
Task dispatch and load balance

Msg. interface:
Message, Synchronization

Computing Core:
Highly optimized for specific library

Local Memory Manage:
Single or Double Buffering

Data Dispatcher

Data Movement:
DMA, Scattering / Gathering

SPU

...
Motivation for Overlays

- Overlays must be used if the sum of the lengths of all the code segments of a program, plus the lengths of the data areas required by the program, exceeds the SPU local storage size.
- They may be used in other circumstances; for example performance might be improved if the size of a data area can be increased by moving rarely used functions (such as error or exception handlers) to overlays.
**Motivation for SPE Software Managed Cache**

- **SPE memory accesses are to Local Store Addresses only. Access to main memory requires explicit DMA calls.**
  - This represents a new programming model

- **Software cache has many benefits in SPE environment**
  - Simplifies programming model
    - familiar load/store effective address model can be used
    - Decreases time to port to SPE
  - Take advantage of locality of reference
  - Can be easily optimized to match data access patterns
Cell BE Basic Programming Concepts
Cell/BE Basic Programming Concepts

- The PPE is just a PowerPC running Linux.
  - No special programming techniques or compilers are needed.
- The PPE manages SPE processes as POSIX pthreads.
- IBM-provided library (libspe2) handles SPE process management within the threads.
- Compiler tools embed SPE executables into PPE executables: one file provides instructions for all execution units.
PPE Programming Environment

- PPE runs PowerPC applications and operating system
- PPE handles thread allocation and resource management among SPEs
- PPE’s Linux kernel controls the SPUs’ execution of programs
  - Schedule SPE execution independent from regular Linux threads
  - Responsible for runtime loading, passing parameters to SPE programs, notification of SPE events and errors, and debugger support
- PPE’s Linux kernel manages virtual memory, including mapping each SPE’s local store (LS) and problem state (PS) into the effective-address space
- The kernel also controls virtual-memory mapping of MFC resources, as well as MFC segment-fault and page-fault handling
- Large pages (16-MB pages, using the hugetlbfs Linux extension) are supported
- Compiler tools embed SPE executables into PPE executables
SPE Programming Environment

- Each SPE has a SIMD instruction set, 128 vector registers and two in-order execution units, and no operating system
- Data must be moved between main memory and the 256 KB of SPE local store with explicit DMA commands
- Standard compilers are provided
  - GNU and XL compilers, C, C++ and Fortran
  - Will compile scalar code into the SIMD-only SPE instruction set
  - Language extensions provide SIMD types and instructions.
- SDK provides math and programming libraries as well as documentation

➤ The programmer must handle

- A set of processors with varied strengths and unequal access to data and communication
- Data layout and SIMD instructions to exploit SIMD utilization
- Local store management (data locality and overlapping communication and computational)
## Cell BE Threads and Tasks Model

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Thread</td>
<td>A thread running on the PPE in the Linux operating-system environment.</td>
</tr>
<tr>
<td>Cell Broadband Engine Linux Task</td>
<td>A task running on the PPE and SPE. Each such task:</td>
</tr>
<tr>
<td></td>
<td>• Has one or more Linux thread and some number of SPE threads.</td>
</tr>
<tr>
<td></td>
<td>• All the Linux threads within the task share the task’s resources, including access to the SPE threads.</td>
</tr>
<tr>
<td>SPE Thread</td>
<td>A thread running on an SPE. Each such thread:</td>
</tr>
<tr>
<td></td>
<td>• Has its own 128 x 128-bit register file, program counter, and MFC Command Queues.</td>
</tr>
<tr>
<td></td>
<td>• Can communicate with other execution units (or with effective-address memory through the MFC channel interface).</td>
</tr>
</tbody>
</table>
PPE Program and SPE Program

- **PPE program** – Linux tasks
  - a Linux task can initiate one or more “SPE threads”

- **SPE program** – “local” SPE executables (“SPE threads”)
  - SPE executables are packaged inside PPE executable files

- **An SPE thread:**
  - is initiated by a task running on the PPE
  - is associated with the initiating task on the PPE
  - runs asynchronously from initiating task
  - has a unique identifier known to both the SPE thread and the initiating task
  - completes at return from main in the SPE code

- **An SPE group:**
  - a collection of SPE threads that share scheduling attributes
  - there is a default group with default attributes
  - each SPE thread belongs to exactly one SPE group
Basic PPE and SPE Program Control and Data Flow

1) (PPE Program) Loads the SPE program to the LS.

2) (PPE Program) Instructs the SPEs to execute the SPE program.

3) (SPE Program) Transfers required data from the main memory to the LS.

4) (SPE Program) Processes the received data in accordance with the requirements.

5) (SPE Program) Transfers the processed result from the LS to the main memory.

6) (SPE Program) Notifies the PPE program of the termination of processing.

Source: PS3-Cell Programming Tutorial
SPE runtime management library

- The SPE runtime management library (libspe) contains an SPE thread programming model for Cell BE applications
- is used to control SPE program execution from the PPE program
- The elfspe enables direct SPE executable execution from a Linux shell without the need for a PPE application creating an SPE thread
- For Cell BE-based blade servers the LIBSPE headers, libraries and binaries are installed by the SDK into the /usr directory and the standalone SPE executive, elfspe, is registered during system root by commands added to /etc/rc.d/init.d
- For the simulator the LIBSPE and ELFSPE binaries and libraries are pre-installed in the same directories in the System Root Image and no further action is required at install time
LIBSPE Version 2.0

- Licensed under the GNU LPGL
- Design is independent of the underlying operating system
  - can run on a variety of OS platforms
- In Linux implementation, it uses of the SPU File System (SPUFS) as
  the Linux kernel interface
- Handles SPEs as virtual objects called SPE contexts
  - SPE programs can be loaded and executed by operating SPE
    contexts
LIBSPE Version 2.0 – *Single thread*

- A simple application uses a single PPE thread, that is, the application’s PPE thread
- The basic scheme for a simple application using an SPE is as follows:
  1. Create an SPE context
  2. Load an SPE executable object into the SPE context’s local store
  3. Run SPE context – this transfers control to the operating system requesting the actual scheduling of the context to a physical SPE in the system
  4. Destroy SPE context
- Note that step 3. above represents a *synchronous call* to the operating system. The calling application’s PPE thread blocks until the SPE stops execution and the operating system returns from the system call invoking the SPE execution
LIBSPE Version 2.0 – Hello World single thread sample – PPE program

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <libspe2.h>

extern spe_program_handle_t hello_spu;

int main(void)
{
    // Load an SPE executable object into the SPE context local store
    if (spe_program_load(speid, &hello_spu) ) {
        perror("spe_program_load");
        return -3;
    }

    // Structure for an SPE context
    spe_context_ptr_t speid;
    unsigned int flags = 0;
    unsigned int entry = SPE_DEFAULT_ENTRY;
    void * argp = NULL;
    void * envp = NULL;
    spe_stop_info_t stop_info;
    int rc;

    // Create an SPE context
    speid = spe_context_create(flags, NULL);
    if (speid == NULL) {
        perror("spe_context_create");
        return -2;
    }

    // Run the SPE context
    rc = spe_context_run(speid, &entry, 0, argp, envp, &stop_info);
    if (rc < 0)
        perror("spe_context_run");

    // Destroy the SPE context
    spe_context_destroy(speid);
    return 0;
}
```
LIBSPE Version 2.0 – Create an SPE Context

// Create an SPE context
    speid = spe_context_create(flags, NULL);
    if (speid == NULL) {
        perror("spe_context_create");
        return -2;
    }

**spe_context_ptr_t spe_context_create(unsigned int flags, spe_gang_context_ptr_t gang)**

*flags*  
A bit-wise OR of modifiers that are applied when the SPE context is created.

*gang*  
Associate the new SPE context with this gang context. If NULL is specified, the new SPE context is not associated with any gang.

On success, a pointer to the newly created SPE context is returned.
LIBSPE Version 2.0 - Load an SPE main program

// Load an SPE executable object into the SPE context local store
if (spe_program_load(speid, &hello_spu) ) {
    perror("spe_program_load");
    return -3;
}

int spe_program_load (spe_context_ptr_t spe, spe_program_handle_t *program)

    spe       A valid pointer to the SPE context for which an SPE program should be loaded.
    program   A valid address of a mapped SPE program

On success, 0 (zero) is returned.

NOTES: spe_program_load loads an SPE main program that has been mapped to memory at the address pointed to by program into the local store of the SPE identified by the SPE context spe. This is mandatory before running the SPE context with spe_context_run.
LIBSPE Version 2.0 – *Destroy the SPE context*

```c
// Destroy the SPE context
spe_context_destroy(speid);
return 0;
```

```c
int spe_context_destroy (spe_context_ptr_t spe)

spe Specifies the SPE context to be destroyed.

On success, 0 (zero) is returned.
```

**NOTES:** *spe_context_destroy* destroys the specified SPE context and free any associated resources.
LIBSPE Version 2.0 – Multi-thread

- Many applications need to use multiple SPEs concurrently
- In this case, it is necessary for the application to create at least as many PPE threads as concurrent SPE contexts are required
- Each of these PPE threads may run a single SPE context at a time
- If N concurrent SPE contexts are needed, it is common to have a main application thread plus N PPE threads dedicated to SPE context execution

The basic scheme for a simple application running N SPE contexts is

1. Create N SPE contexts
2. Load the appropriate SPE executable object into each SPE context’s local store
3. Create N PPE threads
   a. In each of these PPE threads run one of the SPE contexts
   b. Terminate the PPE thread
4. Wait for all N PPE threads to terminate
5. Destroy all N SPE contexts
LIBSPE Version 2.0 – Multi-thread

*Single Thread Asynchronous Execution Example*

- The application creates one PPE thread
- The PPE thread will run an SPE context at a time
- The basic scheme for a simple application running 1 SPE contexts asynchronously is
  1. Create an SPE context
  2. Load the appropriate SPE executable object into the SPE context’s local store
  3. Create a PPE thread
     a. Run the SPE context in the PPE thread
     b. Terminate the PPE thread
  4. Wait for the PPE thread to terminate
  5. Destroy the SPE context
LIBSPE Version 2.0 – Single Thread Asynchronous Execution Example

```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <libib.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t hello_spu;

int main(void)
{
    ppu_thread_data_t data;
    data.context = spe_context_create(0, NULL);
    spe_program_load(data.context, &hello_spu);
    data.entry = SPE_DEFAULT_ENTRY;
    data.flags = 0;
    data.argv = NULL;
    data.envp = NULL;
    pthread_create(&datapthread, NULL, &ppu_thread_function, &data);
    pthread_join(datapthread, NULL);
    spe_context_destroy(data.context);
    return 0;
}
```

// Structure for an SPE thread
type def struct ppu_thread_data{
    spe_context_ptr_t context;
    pthread_t pthread;
    unsigned int entry;
    unsigned int flags;
    void *argv;
    void *envp;
    spe_stop_info_t stopinfo;
} ppu_thread_data_t;

// pthread function to run the SPE context
void *ppu_thread_function(void *arg)
{
    ppu_thread_data_t *datap = (ppu_thread_data_t *)arg;
    int rc;
    rc = spe_context_run(datap->context, &datap->entry, datap->flags, datap->argv, datap->envp, &datap->stopinfo);
    pthread_exit(NULL);
}

Thread header file

Add pthread_t to the SPE context data structure

Wait until the PPE thread terminate

Define a pthread function so we can run the SPE context

Call the ppu_thread_function to run the context in the PPE thread

Define data and fill it with necessary context data and then load SPE program image
LIBSPE Version 2.0 – Multi Thread Asynchronous Execution Example

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <libspe2.h>
#include <pthread.h>

extern spe_program_handle_t simple_spu;

#define SPU_THREADS 6

void *ppu_pthread_function(void *arg) {
    spe_context_ptr_t ctx;
    unsigned int entry = SPE_DEFAULT_ENTRY;

    ctx = *((spe_context_ptr_t*)arg);
    if (spe_context_run(ctx, &entry, 0, NULL, NULL, NULL) < 0) {
        perror("Failed running context");
        exit(1);
    }
    pthread_exit(NULL);
}
LIBSPE Version 2.0 – Multi Thread Asynchronous Execution Example

```c
int main()
{
    int i;
    spe_context_ptr_t ctxs[SPU_THREADS];
    pthread_t threads[SPU_THREADS];

    /* Create several SPE-threads to execute 'simple_spu'. */
    for(i=0; i<SPU_THREADS; i++) {
        /* Create context */
        if ((ctxs[i] = spe_context_create (0, NULL)) == NULL) {
            perror ("Failed creating context");
            exit (1);
        }
        /* Load program into context */
        if (spe_program_load (ctxs[i], &simple_spu)) {
            perror ("Failed loading program");
            exit (1);
        }
    }

    /* Create thread for each SPE context */
    if (pthread_create (&threads[i], NULL, &ppu_thread_function, &ctxs[i])) {
        perror ("Failed creating thread");
        exit (1);
    }

    /* Wait for SPU-thread to complete execution. */
    for (i=0; i<SPU_THREADS; i++) {
        if (pthread_join (threads[i], NULL)) {
            perror ("Failed pthread_join");
            exit (1);
        }
    }

    printf("\nThe program has successfully executed.\n");

    return (0);
}
```
PPE vs SPE

- Both PPE and SPE execute SIMD instructions
  - PPE processes SIMD operations in the VXU within its PPU
  - SPEs process SIMD operations in their SPU
- Both processors execute different instruction sets
- Programs written for the PPE and SPEs must be compiled by different compilers
**PPE and SPE Architectural Differences**

<table>
<thead>
<tr>
<th>Feature</th>
<th>PPE</th>
<th>SPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of SIMD registers</td>
<td>32 (128-bit)</td>
<td>128 (128-bit)</td>
</tr>
<tr>
<td>Organization of register files</td>
<td>separate fixed-point, floating-point, and vector multimedia registers</td>
<td>unified</td>
</tr>
<tr>
<td>Load latency</td>
<td>variable (cache)</td>
<td>fixed</td>
</tr>
<tr>
<td>Addressability</td>
<td>$2^{64}$ bytes</td>
<td>256-KB local store $2^{64}$ bytes DMA</td>
</tr>
<tr>
<td>Instruction set</td>
<td>more orthogonal</td>
<td>optimized for single-precision float</td>
</tr>
<tr>
<td>Single-precision</td>
<td>IEEE 754-1985</td>
<td>extended range</td>
</tr>
<tr>
<td>Doubleword</td>
<td>no doubleword SIMD</td>
<td>double-precision floating-point SIMD</td>
</tr>
</tbody>
</table>
PPE and SPE Language-Extension Differences

- Both operate on 128-bit SIMD vectors
- Only the Vector/SIMD Multimedia Extension instruction set supports pixel vectors
- Only the SPU instruction set supports doubleword vectors

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>PPE</th>
<th>SPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector signed char</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector bool char</td>
<td>x</td>
<td>—</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector signed short</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector bool short</td>
<td>x</td>
<td>—</td>
</tr>
<tr>
<td>vector pixel</td>
<td>x</td>
<td>—</td>
</tr>
<tr>
<td>vector unsigned int</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector signed int</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector bool int</td>
<td>x</td>
<td>—</td>
</tr>
<tr>
<td>vector float</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>vector unsigned long long</td>
<td>—</td>
<td>x</td>
</tr>
<tr>
<td>vector signed long long</td>
<td>—</td>
<td>x</td>
</tr>
<tr>
<td>vector double</td>
<td>—</td>
<td>x</td>
</tr>
</tbody>
</table>
Communication Between the PPE and SPEs

- PPE communicates with SPEs through MMIO registers supported by the MFC of each SPE
- Three primary communication mechanisms between the PPE and SPEs
  - **Mailboxes**
    - Queues for exchanging 32-bit messages
    - Two mailboxes (the SPU Write Outbound Mailbox and the SPU Write Outbound Interrupt Mailbox) are provided for sending messages from the SPE to the PPE
    - One mailbox (the SPU Read Inbound Mailbox) is provided for sending messages to the SPE
  - **Signal notification registers**
    - Each SPE has two 32-bit signal-notification registers, each has a corresponding memory-mapped I/O (MMIO) register into which the signal-notification data is written by the sending processor
    - Signal-notification channels, or *signals*, are inbound (to an SPE) registers
    - They can be used by other SPEs, the PPE, or other devices to send information, such as a buffer-completion synchronization flag, to an SPE
  - **DMAs**
    - To transfer data between main storage and the LS
PPE and SPEs Storage Domains

- Three types of storage domains
  - *main-storage domain*,
  - 8 SPE *local store domains*
  - 8 SPE *channel domains*

- The main-storage domain, which is the entire effective-address space, can be configured by the PPE operating system to be shared by all processors and memory-mapped devices in the system (all I/O is memory-mapped)

- Local-storage and channel problem-state (user-state) domains are private to the SPU, LS, and MFC of each SPE
SPE Local Store Domain

- 256-KB, ECC-protected, single-ported, non-caching memory
- Stores all instructions and data used by the SPU
- Supports one access per cycle from either SPE software or DMA transfers
- SPU instruction prefetches are 128 bytes per cycle
- SPU data-access bandwidth is 16 bytes per cycle, quadword aligned
- DMA-access bandwidth is 128 bytes per cycle
- DMA transfers perform a read-modify-write of LS for writes less than a quadword
- An SPU can only fetch instructions from its own LS with load and store instructions, and it performs no address translation for such accesses
  - With respect to accesses by its SPU, the LS is unprotected and untranslated storage
- An SPE program references its own LS using a Local Store Address (LSA)
LS Access Methods

DMA requests can be sent to an MFC either by software on its associated SPU or on the PPE, or by any other processing device that has access to the MFC's MMIO problem-state registers.

Main storage (effective address space)

<table>
<thead>
<tr>
<th>DMA</th>
<th>Direct Memory Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIB</td>
<td>Element Interconnect Bus</td>
</tr>
<tr>
<td>LS</td>
<td>Local Store</td>
</tr>
<tr>
<td>MFC</td>
<td>Memory Flow Controller</td>
</tr>
<tr>
<td>PPE</td>
<td>PowerPC Processor Element</td>
</tr>
<tr>
<td>SPE</td>
<td>Synergistic Processor Element</td>
</tr>
</tbody>
</table>
# PPE Registers

<table>
<thead>
<tr>
<th>Register Type</th>
<th>Offset</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General-Purpose Registers</strong></td>
<td>0-63</td>
<td>64-bit</td>
<td>Fixed-point instructions operate on the full 64-bit width of the GPRs.</td>
</tr>
<tr>
<td><strong>Floating-Point Registers</strong></td>
<td>0-63</td>
<td>64-bit</td>
<td>32 Floating-Point Registers (FPRs), 64 bits wide. The internal format of floating-point data is the IEEE 754 double-precision format. Single-precision results are maintained internally in the double-precision format.</td>
</tr>
<tr>
<td><strong>Link Register</strong></td>
<td>0-63</td>
<td>64-bit</td>
<td>64-bit LR - to hold the effective address of a branch target.</td>
</tr>
<tr>
<td><strong>Count Register</strong></td>
<td>0-63</td>
<td>64-bit</td>
<td>64-bit CTR - to hold either a loop counter or the effective address of a branch target.</td>
</tr>
<tr>
<td><strong>Fixed-Point Exception Register</strong></td>
<td>0-32</td>
<td>64-bit</td>
<td>64-bit XER - contains the carry and overflow bits and the byte count for the move-assist instructions.</td>
</tr>
<tr>
<td><strong>Condition Register</strong></td>
<td>0-32</td>
<td>64-bit</td>
<td></td>
</tr>
<tr>
<td><strong>Floating-Point Status and Control Register</strong></td>
<td>0-32</td>
<td>64-bit</td>
<td></td>
</tr>
<tr>
<td><strong>Vector Multimedia Registers</strong></td>
<td>0-127</td>
<td>64-bit</td>
<td>32 128-bit-wide VMRs - served as source and destination registers for all vector instructions.</td>
</tr>
<tr>
<td><strong>Vector Status and Control Register</strong></td>
<td>0-32</td>
<td>64-bit</td>
<td></td>
</tr>
<tr>
<td><strong>VR Save/Restore Register</strong></td>
<td>0-32</td>
<td>64-bit</td>
<td></td>
</tr>
</tbody>
</table>
PPE VMX Instructions

- VMX instructions are 4 bytes long and word-aligned
- support simultaneous execution on multiple elements that make up the 128-bit vector operands
- vector elements may be byte, halfword, or word
- The 128-bit Vector/SIMD Multimedia Extension unit (VXU) operates concurrently with the PPU's fixed-point integer unit (FXU) and floating-point execution unit (FPU)
Example of a VMX Program

#include <stdio.h>
// Define a type we can look at either as an array of ints or as a vector.
typedef union {
    int iVals[4];
    vector signed int myVec;
} vecVar;

int main()
{
    vecVar v1, v2, vConst; // define variables
    // load the literal value 2 into the 4 positions in vConst,
    vConst.myVec = (vector signed int){2, 2, 2, 2};
    // load 4 values into the 4 element of vector v1
    v1.myVec = (vector signed int){10, 20, 30, 40};
    // call vector add function
    v2.myVec = vec_add(v1.myVec, vConst.myVec);
    // see what we got!
           v2.iVals[0], v2.iVals[1], v2.iVals[2], v2.iVals[3]);
    return 0;
}
SPE Registers

128 of 128-bit General-Purpose Registers (GPRs) that can be used to store all data types

The Floating-Point Status and Control Register (FPSCR) records information about the result and any associated exceptions.
SPE Local Store

- Holds instructions and data
- Filled with instructions and data using DMA transfers initiated from SPU or PPE software
- DMA operations are buffered and can only access the LS at most one of every eight cycles
- Instruction prefetches deliver at least 17 instructions sequentially from the branch target
  - impact of DMA operations on SPU loads and stores and program-execution times is, by design, limited.
SPU Instruction Set

- Operates primarily on SIMD vector operands, both fixed-point and floating-point,
- Supports some scalar operands
- Supports *big-endian* data ordering,
  - lowest-address byte and lowest-numbered bit are the most-significant (high) byte and bit
- Supports data types
  - byte—8 bits
  - halfword—16 bits
  - word—32 bits
  - doubleword—64 bits
  - quadword—128 bits

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>Sixteen 8-bit unsigned chars</td>
</tr>
<tr>
<td>vector signed char</td>
<td>Sixteen 8-bit signed chars</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>Eight 16-bit unsigned halfwords</td>
</tr>
<tr>
<td>vector signed short</td>
<td>Eight 16-bit signed halfwords</td>
</tr>
<tr>
<td>vector unsigned int</td>
<td>Four 32-bit unsigned words</td>
</tr>
<tr>
<td>vector signed int</td>
<td>Four 32-bit signed words</td>
</tr>
<tr>
<td>vector unsigned long long</td>
<td>Two 64-bit unsigned doublewords</td>
</tr>
<tr>
<td>vector signed long long</td>
<td>Two 64-bit signed doublewords</td>
</tr>
<tr>
<td>vector float</td>
<td>Four 32-bit single-precision floats</td>
</tr>
<tr>
<td>vector double</td>
<td>Two 64-bit double precision floats</td>
</tr>
<tr>
<td>quadword</td>
<td>quadword (16-byte)</td>
</tr>
</tbody>
</table>
SPU Instruction Types

204 instructions grouped into 11 groups

<table>
<thead>
<tr>
<th>Type</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Load and Store</td>
<td>16</td>
</tr>
<tr>
<td>Constant Formation</td>
<td>6</td>
</tr>
<tr>
<td>Integer and Logical Operations</td>
<td>59</td>
</tr>
<tr>
<td>Shift and Rotate</td>
<td>31</td>
</tr>
<tr>
<td>Compare, Branch, and Halt</td>
<td>40</td>
</tr>
<tr>
<td>Hint-For-Branch</td>
<td>3</td>
</tr>
<tr>
<td>Floating-Point</td>
<td>28</td>
</tr>
<tr>
<td>Control</td>
<td>8</td>
</tr>
<tr>
<td>SPU Channel</td>
<td>3</td>
</tr>
<tr>
<td>SPU Interrupt Facility</td>
<td>7</td>
</tr>
<tr>
<td>Synchronization and Ordering</td>
<td>3</td>
</tr>
</tbody>
</table>

All computational instructions operate on registers—there are no computational instructions that modify storage ➔ no need for LS cache
Register Layout of Data Types and Preferred Slot

When instructions use or produce scalar operands or addresses, the values are in the preferred scalar slot.

The left-most word (bytes 0, 1, 2, and 3) of a register is called the **preferred slot**.

<table>
<thead>
<tr>
<th>Preferred Slot</th>
<th>Preferred Scalar Slot:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Byte</td>
<td></td>
</tr>
<tr>
<td>Halfword</td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>Doubleword</td>
<td></td>
</tr>
<tr>
<td>Quadword</td>
<td></td>
</tr>
</tbody>
</table>
Promoting Scalar Data Types to Vector Data Types

- SPU only loads and stores a quadword at a time
- Value of scalar operands (including addresses) is kept in the preferred slot of a SIMD register
- Scalar (sub quadword) loads and stores require several instructions to format the data for use on the SIMD architecture of the SPE
  - e.g., scalar stores require a read, scalar insert, and write operation
- Strategies to make operations on scalar data more efficient:
  - Change the scalars to quadword vectors to eliminate three extra instructions associated with loading and storing scalars
  - Cluster scalars into groups, and load multiple scalars at a time using a quadword memory access. Manually extract or insert the scalars as needed. This will eliminate redundant loads and stores.

*Intrinsics for Changing Scalar and Vector Data Types*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = spu_insert</td>
<td>Insert a scalar into a specified vector element.</td>
</tr>
<tr>
<td>d = spu_promote</td>
<td>Promote a scalar to a vector.</td>
</tr>
<tr>
<td>d = spu_extract</td>
<td>Extract a vector element from its vector.</td>
</tr>
</tbody>
</table>
#include <stdio.h>

// Define a type we can look at either as an array of ints or as a vector.
typedef union {
    int iVals[4];
    vector signed int myVec;
} vecVar;

int main()
{
    vecVar v1, v2, vConst; // define variables
    // load the literal value 2 into the 4 positions in vConst,
    vConst.myVec = (vector signed int){2, 2, 2, 2};
    // load 4 values into the 4 element of vector v1
    v1.myVec = (vector signed int){10, 20, 30, 40};
    // call vector add function
    v2.myVec = spu_add( v1.myVec, vConst.myVec );
    // see what we got!
    printf("\nResults:\nv2[0] = %d, v2[1] = %d, v2[2] = %d, v2[3] = %d\n\n",
        v2.iVals[0], v2.iVals[1], v2.iVals[2], v2.iVals[3]);
    return 0;
}
Developing Code for Cell – DMA & Mailboxes
Cell’s Primary Communication Mechanisms

- DMA transfers, mailbox messages, and signal-notification
- All three are implemented and controlled by the SPE’s MFC

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA transfers</td>
<td>Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.</td>
</tr>
<tr>
<td>Mailboxes</td>
<td>Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.</td>
</tr>
<tr>
<td>Signal notification</td>
<td>Used for control communication from the PPE or other devices. Signal notification (also called signaling) uses 32-bit registers that can be configured for one-sender-to-one-receiver signaling or many-senders-to-one-receiver signaling.</td>
</tr>
</tbody>
</table>
DMA Commands

- MFC commands that transfer data are referred to as DMA commands
- Transfer direction for DMA commands referenced from the SPE
  - Into an SPE (from main storage to local store) → **get**
  - Out of an SPE (from local store to main storage) → **put**
DMA Commands

Channel Control
Intrinsics

\textit{spu\_writetch}

Composite
Intrinsics

\textit{spu\_mfcdma32}

MFC Commands

\textit{mfc\_get}

defined as macros in
\textit{spu\_mfcio.h}

For details see: SPU C/C++ Language Extensions
DMA Get and Put Command (SPU)

- **DMA get from main memory into local store**
  
  (void) mfc_get(volatile void *ls, uint64_t ea, uint32_t size,
  uint32_t tag, uint32_t tid, uint32_t rid)

- **DMA put into main memory from local store**
  
  (void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size,
  uint32_t tag, uint32_t tid, uint32_t rid)

- **To ensure order of DMA request execution:**
  
  - mfc_putf : **fenced** (all commands executed before within the same tag group must finish first, later ones could be before)

  - mfc_putb : **barrier** (the barrier command and all commands issued thereafter are not executed until all previously issued commands in the same tag group have been performed)
DMA-Command Tag Groups

- 5-bit DMA Tag for all DMA commands (except getllar, putllc, and putlluc)
- Tag can be used to
  - determine status for entire group or command
  - check or wait on the completion of all queued commands in one or more tag groups
- Tagging is optional but can be useful when using barriers to control the ordering of MFC commands within a single command queue.
- Synchronization of DMA commands within a tag group: fence and barrier
  - Execution of a fenced command option is delayed until all previously issued commands within the same tag group have been performed.
  - Execution of a barrier command option and all subsequent commands is delayed until all previously issued commands in the same tag group have been performed.
Barriers and Fences

![Diagram showing barriers and fences in execution slots with time and instructions]

- **Barrier**: Indicates a synchronization point where all processes must wait before proceeding.
- **Fence**: Similar to a barrier, but may have different implications in terms of synchronization and ordering of instructions.

Legend:
- **Synchronizing command**
- **Non-synchronizing command**
- **Execution slot**
DMA Characteristics

- **DMA transfers**
  - transfer sizes can be 1, 2, 4, 8, and n*16 bytes (n integer)
  - maximum is 16KB per DMA transfer
  - 128B alignment is preferable

- **DMA command queues per SPU**
  - 16-element queue for SPU-initiated requests
  - 8-element queue for PPE-initiated requests
  - SPU-initiated DMA is always preferable

- **DMA tags**
  - each DMA command is tagged with a 5-bit identifier
  - same identifier can be used for multiple commands
  - tags used for polling status or waiting on completion of DMA commands

- **DMA lists**
  - a single DMA command can cause execution of a list of transfer requests (in LS)
  - lists implement scatter-gather functions
  - a list can contain up to 2K transfer requests
Transfer from PPE (Main Memory) to SPE

- **DMA get from main memory**
  
  \[
  \text{mfc\_get}(\text{lsaddr}, \text{ea}, \text{size}, \text{tag\_id}, \text{tid}, \text{rid});
  \]
  
  - \text{lsaddr} = target address in SPU local store for fetched data (SPU local address)
  - \text{ea} = effective address from which data is fetched (global address)
  - \text{size} = transfer size in bytes
  - \text{tag\_id} = tag-group identifier
  - \text{tid} = transfer-class id
  - \text{rid} = replacement-class id

- **Also available via “composite intrinsic”:**

  \[
  \text{spu\_mfcdma64}(\text{lsaddr}, \text{eahi}, \text{ealow}, \text{size}, \text{tag\_id}, \text{cmd});
  \]
DMA Command Status (SPE)

- DMA read and write commands are non-blocking
- Tags, tag groups, and tag masks used for:
  - checking status of DMA commands
  - waiting for completion of DMA commands
- Each DMA command has a 5-bit tag
  - commands with same tag value form a “tag group”
- Tag mask is used to identify tag groups for status checks
  - tag mask is a 32-bit word
  - each bit in the tag mask corresponds to a specific tag id:

\[
tag\_mask = (1 \ll tag\_id)\]
DMA Tag Status (SPE)

- **Set tag mask**
  
  ```c
  unsigned int tag_mask;
  mfc_write_tag_mask(tag_mask);
  - tag mask remains set until changed
  ```

- **Fetch tag status**
  
  ```c
  unsigned int result;
  result = mfc_read_tag_status();  /* or mfc_stat_tag_status(); */
  - tag status is logically ANDed with current tag mask
  - tag status bit of ‘1’ indicates that no DMA requests tagged with the specific tag id (corresponding to the status bit location) are still either in progress or in the DMA queue
  ```
Waiting for DMA Completion (SPE)

- **Wait for any tagged DMA:**
  
  ```c
  mfc_read_tag_status_any();
  ```
  
  - wait until **any** of the specified tagged DMA commands is completed

- **Wait for all tagged DMA:**

  ```c
  mfc_read_tag_status_all();
  ```
  
  - wait until **all** of the specified tagged DMA commands are completed

  ➢ **Specified tagged DMA commands = command specified by current tag mask setting**
DMA Example: Read into Local Store

```c
inline void dma_mem_to_ls(unsigned int mem_addr,
    volatile void *ls_addr, unsigned int size)
{
    unsigned int tag = 0;
    unsigned int mask = 1;
    mfc_get(ls_addr, mem_addr, size, tag, 0, 0);
    mfc_write_tag_mask(mask);
    mfc_read_tag_status_all();
}
```

- Read contents of mem_addr into ls_addr
- Set tag mask
- Wait for all tag DMA completed
DMA Example: Write to Main Memory

```c
inline void dma_ls_to_mem(unsigned int mem_addr, volatile void *ls_addr, unsigned int size)
{
    unsigned int tag = 0;
    unsigned int mask = 1;
    mfc_put(ls_addr, mem_addr, size, tag, 0, 0);
    mfc_write_tag_mask(mask);
    mfc_read_tag_status_all();
}
```

- **Write contents of mem_addr into ls_addr**
- **Set tag mask**
- **Wait for all tag DMA completed**
SPE – SPE DMA

- Address in the other SPE’s local store is represented as a 32-bit effective address (global address)
- SPE issuing the DMA command needs a pointer to the other SPE’s local store as a 32-bit effective address (global address)
- PPE code can obtain effective address of an SPE’s local store:

  ```c
  #include <libspe.h>
  speid_t speid;
  void *spe_ls_addr;
  ...
  spe_ls_addr = spe_get_ls(speid);
  ```

- Effective address of an SPE’s local store can then be made available to other SPEs (e.g. via DMA or mailbox)
Tips to Achieve Peak Bandwidth for DMAs

- The performance of a DMA data transfer is best when the source and destination addresses have the same quadword offsets within a PPE cache line.
- Quadword-offset-aligned data transfers generate full cache-line bus requests for every unrolling, except possibly the first and last unrolling.
- Transfers that start or end in the middle of a cache line transfer a partial cache line (less than 8 quadwords) in the first or last bus request, respectively.
Mailboxes Overview
Uses of Mailboxes

- To communicate messages up to 32 bits in length, such as buffer completion flags or program status
  - e.g., When the SPE places computational results in main storage via DMA. After requesting the DMA transfer, the SPE waits for the DMA transfer to complete and then writes to an outbound mailbox to notify the PPE that its computation is complete

- Can be used for any short-data transfer purpose, such as sending of storage addresses, function parameters, command parameters, and state-machine parameters

- Can also be used for communication between an SPE and other SPEs, processors, or devices
  - Privileged software needs to allow one SPE to access the mailbox register in another SPE by mapping the target SPE’s problem-state area into the EA space of the source SPE. If software does not allow this, then only atomic operations and signal notifications are available for SPE-to-SPE communication.
Mailboxes - Characteristics

Each MFC provides three mailbox queues of 32 bit each:

1. PPE (“SPU write outbound”) mailbox queue
   - SPE writes, PPE reads
   - 1 deep
   - SPE stalls writing to full mailbox

2. PPE (“SPU write outbound”) interrupt mailbox queue
   - like PPE mailbox queue, but an interrupt is posted to the PPE when the mailbox is written

3. SPU (“SPU read inbound”) mailbox queue
   - PPE writes, SPE reads
   - 4 deep
   - can be overwritten

➢ Each mailbox entry is a fullword
Special Notices -- Trademarks

This document was developed for IBM offerings in the United States as of the date of publication. IBM may not make these offerings available in other countries, and the information is subject to change without notice. Consult your local IBM business contact for information on the IBM offerings available in your area. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1785 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided "AS IS" with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

IBM Global Financing offerings are provided through IBM Credit Corporation in the United States and other IBM subsidiaries and divisions worldwide to qualified commercial and government clients. Rates are based on a client's credit rating, financing terms, offering type, equipment type and options, and may vary by country. Other restrictions may apply. Rates and offerings are subject to change, extension or withdrawal without notice.

IBM is not responsible for printing errors in this document that result in pricing or information inaccuracies.

All prices shown are IBM's United States suggested list prices and are subject to change without notice; reseller prices may vary.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Many of the features described in this document are operating system dependent and may not be available on Linux. For more information, please check: http://www.ibm.com/systems/p/software/whitepapers/linux_overview.html

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.
Special Notices (Cont.) -- Trademarks


A full list of U.S. trademarks owned by IBM may be found at: http://www.ibm.com/legal/copytrade.shtml.

Cell Broadband Engine and Cell Broadband Engine Architecture are trademarks of Sony Computer Entertainment, Inc. in the United States, other countries, or both.

Rambus is a registered trademark of Rambus, Inc.
XDR and FlexIO are trademarks of Rambus, Inc.

UNIX is a registered trademark in the United States, other countries or both.

Linux is a trademark of Linus Torvalds in the United States, other countries or both.

Fedora is a trademark of Redhat, Inc.

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries or both.

Intel, Intel Xeon, Itanium and Pentium are trademarks or registered trademarks of Intel Corporation in the United States and/or other countries.

AMD Opteron is a trademark of Advanced Micro Devices, Inc.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

TPC-C and TPC-H are trademarks of the Transaction Performance Processing Council (TPPC).

SPECint, SPECfp, SPECjbb, SPECweb, SPECjAppServer, SPEC OMP, SPECviewperf, SPECapc, SPEChpc, SPECjvm, SPECmail, SPECimap and SPECsfs are trademarks of the Standard Performance Evaluation Corp (SPEC).

Altivec is a trademark of Freescale Semiconductor, Inc.

PCI-X and PCI Express are registered trademarks of PCI SIG.

InfiniBand™ is a trademark the InfiniBand® Trade Association

Other company, product and service names may be trademarks or service marks of others.

Revised July 23, 2006
Special Notices - Copyrights


The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.

IBM IBM Logo Power Architecture

Other company, product and service names may be trademarks or service marks of others.

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Microelectronics Division
1580 Route 52, Bldg. 504
Hopewell Junction, NY 12533-6351

The IBM home page is http://www.ibm.com
The IBM Microelectronics Division home page is http://www.chips.ibm.com