Cell Software Solutions Programming Model

Duc Vianney, Ph. D.
Cell Ecosystem & Solutions Development, IBM Corporation
Acknowledgments

- Peter Hofstee, Ph.D., STIDC
- Dan Brokenshire, STIDC
- Jeff Derby, Ph.D., E&TS Performance Analysis
- Michael Perrone, Ph.D., Cell Solutions Development, IBM Research
- among many others ….

Trademarks

Cell Broadband Engine ™ is a trademark of Sony Computer Entertainment, Inc.
Outline

- Cell overview
- **Cell hardware/software solutions challenges**
- **Cell programming models**
  - PPE-centric vs. SPE-centric
  - PPE programming models
  - SPE programming models
  - Function offload
  - Overlapping DMA and computation
  - Heterogeneous multi-thread
- **Cell software solutions development guidelines**
- **Cell performance tips and practices**
- **Cell Ideal & non-ideal software**
Cell Overview

- **CBE or “Cell Broadband Engine”**
  - Also known as: BE, Cell processor

- **CBE includes:**
  - PPC core with “traditional” memory subsystem
  - 8 “synergistic processing elements”
  - very high bandwidth internal interconnect “Element Interconnect Bus” (BE data ring)
  - I/O interfaces (2)
Cell Overview – system statistics

- Cell includes 1PPE + 8SPEs
  - provided more than 8x compute capability than traditional processors
    - de-coupled SIMD engines for growth and scalability
  - simple PPC micro-architecture
    - in-order; dual issue
    - dual-thread support
    - minimal chip area / power budget
- 8 SPU SIMD engines provide tremendous compute power
  - in-order; dual issue
  - dedicated resources
    - 128 128-bit registers
    - 256KB local store
    - 2x25.6GB/s DMA, etc.
  - up to 16-way SIMD for exploiting data parallelism
    - 25.6 SP GFlops/1.6 DP GFlops per SPU
- EIB (BE data ring) for intra-BE and external communication
  - 204.8GB/s peak bandwidth
  - 25.6GB/s memory b/w
  - 35GB/s (out)+25GB/s (in) IO
Cell Broadband Engine Processor

L2 (512KB)
Test & Debug Logic
Element Interconnect Bus
Cell Solutions Challenges

- **Hardware challenges**
  - Small Main Memory per chip (512MB per Cell Processor) and small local memory store per SPE
  - Single Precision floating point uses truncated rounding (decreased accuracy in favor of speed)

- **Software challenges**
  - Operating System requires patches
  - Programming model & non-homogeneous multiprocessing
    - Limited set of optimized libraries; tool chain in progress, no commercial-level Fortran
  - Enablement/Integration with existing products
    - Middleware, commercial libraries, GPFS, management

- **Solutions design challenges**
  - Two Levels of Parallelism
    - Regular vector data that is SIMD-able
    - Independent tasks that may be executed in parallel
  - Computational
    - SIMD engines on 8 SPEs and 1 PPE (multi-threaded)
    - Parallel sequence to be distributed over 8 SPE / 1 PPE
    - 256KB local store per SPE usage (data + code)
  - Communicational
    - DMA and Bus bandwidth
      - DMA granularity – 128 bytes
      - DMA bandwidth among LS and System memory, and DMA latency
    - Traffic control
      - Exploit computational complexity and data locality to lower data traffic requirement
    - Shared memory / Message passing abstraction overhead
    - Synchronization
The Need for Cell Programming Model

- Cell provides a massive computational capacity.
- Cell provides a huge communicational bandwidth.
- The resources are distributed.
- A properly selected Cell programming model provides a programmer a systematic and cost-effective framework to apply Cell resources to a particular class of applications.
- A Cell programming model may be supported by language constructs, runtime, libraries, or object-oriented frameworks.
Programming Model Influencers

- PowerPC 64 compliant
- High speed coherent interconnect
- Limited local store size
- Coherent shared memory
- Signal Notification Registers
- Large SPE context
- Atomic operations
- Multiple execution units
- Aliased LS memory
- Mailboxes
- Heterogeneous
- Multi-threading
- SIMD
- SPE Events
- Bandwidth Reservations
- SW managed DMA engines
- High speed EIB
- VM address translation and protection
- DMA list supporting scatter / gather
- Direct problem state mapping
- Resource Management Tables
- DMA alignment & size restrictions
Programming Models

- One focus is on how an application can be partitioned across the processing elements
  - PPE, SPEs

- Partitioning involves consideration of and trade-offs among:
  - processing load
  - program structure
  - data flow
  - data and code movement via DMA
  - loading of bus and bus attachments
  - desired performance

- Several models:
  - “PPE-centric” vs. “SPE-centric”
  - “data-serial” vs. “data-parallel”
  - others …
“PPE-Centric” & “SPE-Centric” Models

- **“PPE-Centric”:**
  - an offload model
  - main line application code runs in PPC core
  - individual functions extracted and offloaded to SPEs
  - SPUs wait to be given work by the PPC core

- **“SPE-Centric”:**
  - most of the application code distributed among SPEs
  - PPC core runs little more than a resource manager for the SPEs (e.g. maintaining in main memory control blocks with work lists for the SPEs)
  - SPE fetches next work item (what function to execute, pointer to data, etc.) from main memory (or its own memory) when it completes current work item
Single Cell Programming Models

Single Cell environment:
- PPE programming models
- SPE Programming models
  - Small single-SPE models
  - Large single-SPE models
  - Multi-SPE parallel programming models
- Cell Embedded SPE Object Format (CESOF)
PPE Programming Model (participation)

- PPE is a 64-bit PowerPC core, hosting operating systems and hypervisor
- PPE program inherits traditional programming models
- Cell environment: a PPE program serves as a controller or facilitator
  - CESOF support provides SPE image handles to the PPE runtime
  - PPE program establishes a runtime environment for SPE programs
    - e.g. memory mapping, exception handling, SPE run control
  - It allocates and manages Cell system resources
    - SPE scheduling, hypervisor CBEA resource management
  - It provides OS services to SPE programs and threads
    - e.g. printf, file I/O
Small Single-SPE Models

- Single tasked environment
- Small enough to fit into a 256KB-local store
- Sufficient for many dedicated workloads
- Separated SPE and PPE address spaces – LS / EA
- Explicit input and output of the SPE program
  - Program arguments and exit code per SPE ABI
  - DMA
  - Mailboxes
  - SPE side system calls
- Foundation for a function offload model or a synchronous RPC model
  - Facilitated by interface description language (IDL)
Small Single-SPE Models – tools and environment

- SPE compiler/linker compiles and links an SPE executable
- The SPE executable image is embedded as reference-able RO data in the PPE executable (CESOF)
- A Cell programmer controls an SPE program via a PPE controlling process and its SPE management library
  - i.e. loads, initializes, starts/stops an SPE program
- The PPE controlling process, OS/PPE, and runtime/(PPE or SPE) together establish the SPE runtime environment, e.g. argument passing, memory mapping, system call service.
Large Single-SPE Programming Models

- Data or code working set cannot fit completely into a local store
- The PPE controlling process, kernel, and libspe runtime set up the system memory mapping as SPE’s secondary memory store
- The SPE program accesses the secondary memory store via its software-controlled SPE DMA engine - Memory Flow Controller (MFC)
Large Single-SPE Programming Models – I/O data

- **System memory for large size input / output data**
  - e.g. Streaming model

```plaintext
int g_ip[512*1024]

System memory

int g_op[512*1024]
```

- SPE program: `op = func(ip)`

- Local store

- DMA
Large Single-SPE Programming Models

- **System memory as secondary memory store**
  - Manual management of data buffers
  - Automatic software-managed data cache
    - Software cache framework libraries
    - Compiler runtime support
Large Single-SPE Programming Models

- **System memory as secondary memory store**
  - Manual loading of plug-in into code buffer
    - Plug-in framework libraries
  - Automatic software-managed code overlay
    - Compiler generated overlaying code

![Diagram showing the memory layout with local store and system memory, including SPE plug-in a, b, c, d, e, and f.]
Large Single-SPE Prog. Models – Job Queue

- Code and data packaged together as inputs to an SPE kernel program
- A multi-tasking model (more discussion later)

Diagram:
- Local store
  - Code n
  - Data n
  - SPE kernel
- System memory
  - code/data ...
  - Job queue
  - code/data n
  - code/data n+1
  - code/data n+2
- DMA
Large Single-SPE Programming Models - DMA

- DMA latency handling is critical to overall performance for SPE programs moving large data or code
- Data pre-fetching is a key technique to hide DMA latency – e.g. double-buffering
Large Single-SPE Programming Models - CESOF

- **Cell Embedded SPE Object Format (CESOF)** and PPE/SPE toolchains support the resolution of SPE references to the global system memory objects in the effective-address space.

---

*Diagram details:*
- Local Store Space
  - `_EAR_g_foo` structure
  - `Char local_foo[512]`
- Effective Address Space
  - CESOF EAR symbol resolution
  - DMA transactions
  - `Char g_foo[512]`
Parallel Programming Models

- **Traditional parallel programming models applicable**
- **Based on interacting single-SPE programs**
- **Parallel SPE program synchronization mechanism**
  - Cache line-based MFC atomic update commands similar to the PowerPC lwarx, ldarx, stwcx, and stdcx instructions
  - SPE input and output mailboxes with PPE
  - SPE signal notification / register
  - SPE events and interrupts
  - SPE busy poll of shared memory location
Parallel Programming Models – Shared Memory

- **Access data by address**
  - Random access in nature
- **CESOF support for shared effective-address variables**
- **With proper locking mechanism, large SPE programs may access shared memory objects located in the effective-address space**
- **Compiler OpenMP support**
Parallel Programming Models – Shared Memory

- Exploit standalone SPE programs – “SPUlet”
- Multiple SPE cooperative computing through shared system memory.
- PPE assists in OS services
  - File and network I/O, shared memory access, etc…

Diagram:
- PPE
- SPE 0
- SPE 1
- …
- SPE n
- Shared System Memory

Connections:
- OS Request
- Data Access
Parallel Programming Models – Streaming

- Large array of data fed through a group of SPE programs
- A special case of job queue with regular data
- Each SPE program locks on the shared job queue to obtain next job
- For uneven jobs, workloads are self-balanced among available SPEs
Parallel Programming Models – Message Passing

- **Access data by connection**
  - Sequential in nature

- **Applicable to SPE programs where addressable data space only spans over local store**

- **The message connection is still built on top of the shared memory model**

- **Compared with software-cache shared memory model**
  - More efficient runtime is possible, no address info handling overhead once connected
  - LS to LS DMA optimized for data streaming through pipeline model
Parallel Programming Models – Pipeline

- Use LS to LS DMA bandwidth, not system memory bandwidth
- Flexibility in connecting pipeline functions
- Larger collective code size per pipeline
- Load-balance is harder
Multi-tasking SPEs – LS resident multi-tasking

- Simplest multi-tasking programming model
- No memory protection among tasks
- Co-operative, Non-preemptive, event-driven scheduling

![Diagram showing multi-tasking SPEs]

- Event Queue
  - Task a
  - Task b
  - Task c
  - Task d
  - Task x

- Event Dispatcher

- Local Store

- SPE n
Multi-tasking SPEs – Self-managed multi-tasking

- Non-LS resident
- Blocked job context is swapped out of LS and scheduled back later to the job queue once unblocked
Multi-tasking SPEs – Kernel managed

- **Kernel-level SPE management model**
  - SPE as a device resource
  - SPE as a heterogeneous processor
  - SPE resource represented as a file system

- **SPE scheduling and virtualization**
  - Maps running threads over a physical SPE or a group of SPEs
  - More concurrent logical SPE tasks than the number of physical SPEs
  - High context save/restore overhead
    - favors run-to-completion scheduling policy
  - Supports pre-emptive scheduling when needed
  - Supports memory protection
“data-serial” vs. “data-parallel” Models

- Function offload
  - Application accelerator model
- Application specific accelerators
- Heterogeneous multi-threading
Function Offload

- **Dedicated Function (problem/privileged subsystem)**
  - Programmer writes/uses SPU enabled "libraries"
    - Graphics Pipeline
    - Audio Processing
    - MPEG Encoding/Decoding
    - Encryption / Decryption
  - Main Application in PPE, invokes SPU bound services
    - RPC Like Function Call
    - I/O Device Like Interface (FIFO/ Command Queue)
  - 1 or more SPUs cooperating in subsystem
    - Problem State (Application Allocated)
    - Privileged State (OS Allocated)
  - Code-to-data or data-to-code pipelining possible
  - Very efficient in real-time data streaming applications
A Pipelined Model

- Data-serial
- Example: three function groups, so three SPEs
- Dataflow is unidirectional
- Synchronization is important
  - time spent in each function group should be about the same
  - but may complicate tuning and optimization of code
- Main data movement is SPE-to-SPE
  - can be push or pull
A Data-Partitioned Model

- Data-parallel
- Example: data blocks partitioned into three sub-blocks, so three SPEs
- May require coordination among SPEs between functions
  - e.g. if there is interaction between data sub-blocks
- Essentially all data movement is SPE-to-main memory or main memory-to-SPE

Function 0 in each SPE
- then -
Function 1 in each SPE
- then -
Function 2 in each SPE
- etc.
Function Offload

Application Specific Accelerators

- Acceleration provided by OS or application libraries
- Application portability maintained with platform specific libraries
Overlapping DMA and Computation Model

- **DMA transactions see latency in addition to transfer time**
  - e.g. SPE DMA get from main memory may see a 475-cycle latency

- **Double (or multiple) buffering of data can hide DMA latencies under computation**, e.g. the following is done simultaneously:
  - process current input buffer and write output to current output buffer in SPE LS
  - DMA next input buffer from main memory
  - DMA previous output buffer to main memory
  - requires blocking of inner loops

- **Trade-offs because SPE LS is relatively small**
  - double buffering consumes more LS
  - single buffering has a performance impact due to DMA latency

```plaintext
for i=0 to n-1
  Initiate DMA 0
  Initiate DMA i+1

  Wait for DMA i
  Compute on i
  Wait for DMA n
  Compute on n
```
Heterogeneous Multi-Thread Model

- **Current Linux Operating System Runtime Strategy**
  - PPE Threads, SPE Threads
  - SPE DMA EA = PPE Process EA Space
  - OS supports Create/Destroy SPE tasks
  - Atomic Update Primitives used for Mutex
  - SPE Context Fully Managed
    - Context Save/Restore for Debug
    - Virtualization Mode (indirect access)
    - Direct Access Mode (realtime)
  - OS assignment of SPE threads to SPEs
    - Programmer directed using affinity mask & NUMA topological services

- PPE - control processor
- SPE - data plane processor

- **Application Source & Libraries**
  - PPE object files
  - SPE object files
Cell Software Solutions Development Guidelines

- Algorithm complexity study
- Data layout/locality and Data flow analysis
- Experimental partitioning and mapping of the algorithm and program structure to the architecture
- Develop PPE Control, PPE Scalar code
- Develop PPE Control, partitioned SPE scalar code
  - Communication, synchronization, latency handling
- Transform SPE scalar code to SPE SIMD code
- Re-balance the computation / data movement
- Other optimization considerations
  - PPE SIMD, system bottle-neck, load balance
Application Algorithm Types (Cell Affinity)

- **Pair and Sequence comparisons**
  - Examples: Rich Media Mining, Bioinformatics, SPAM filtering, monitoring, surveillance

- **Data transformation**
  - Examples: Transcoding (e.g. Mpeg to 2 Mpeg 4), XSLT, repurposing, Affine transforms (graphics), encryption/compression, decryption/decompression, Video compression/transformation, visualization

- **Computation**
  - Examples: Ray Tracing, low precision/Game Physics, Matrix multiply, FFT
  - Any computations that can be reliably done in SP FP
  - Data Parallel Floating Point (SIMD)
    - Examples: FSS (Monte Carlo), DSP Algorithms, Ray Tracing, FFT
Code Performance and Optimization

- Many factors affect code performance
  - and most if not all of these are under the control of the programmer

- Start at the beginning: application partitioning strategy
  - some approaches and trade-offs may be difficult to see a priori
  - but there are guidelines that can be followed

- Certain items should always be kept in mind when performance is important
  - data alignment issues
  - instruction-set characteristics
  - microarchitecture characteristics

- What follows is a list of tips and practices
Performance Tips & Practices

- **Application partitioning**
  - investigate different alternatives
  - offload most work to SPE. In general, the greater the percentage of total code that runs in SPEs instead of in the PPE, the better (“SPE-centric” rather than “PPE-centric”)
  - allocate work that are autonomous and non-synchronized
  - pay attention to data type differences between PPE and SPE

- **DMA**
  - use SPE-initiated DMAs rather than PPE-initiated DMAs
  - use multiple-buffering in SPEs to hide DMA latencies under useful processing
  - use cache-line (128B) alignment of blocks being transferred whenever possible

- **PPE code management**
  - PPE code can include software-directed prefetching (e.g. ‘dcbt’ instruction – prefetch a single cache line from memory to both L1 and L2 caches)
  - avoid PPE pre-accesses to large datasets intended primarily for use by SPEs (so that SPE-initiated DMAs transfer from main memory and not from PPE’s L2)
  - exploit PPE multi-threading capability
  - allocate large data sets from large pages to reduce TLB thrashing. The CBE supports 3 concurrent page sizes: 4KB, and 2 from 64KB, 1MB and 16MB.
More Performance Tips & Practices

- **SIMD**
  - investigate different vectorization alternatives
  - take advantage of data reorganization within the register file (which can often occur in parallel with computation)
  - use ‘vector select’ instruction to deal with some ‘if-then-else’ constructs

- **Data Structures**
  - design data structures for efficient SPE data accesses.
  - be aware of data alignment, access patterns, and location
  - MFC supports transfer of 1,2,4,8,n*16 (up to 16K) bytes. Transfer less than 16B must be naturally aligned to minimize EIB bus utilization

- **SPE inner loops**
  - use unrolling and software pipelining where possible
  - keep issue rules in mind
  - be aware that sometimes increasing the number of instructions in an inner loop can actually reduce the cycle count per pass through the loop
  - use the SPE static timing analysis tool (see below)
Still More Performance Tips & Practices

- **SPE branch behavior**
  - make sure ‘branch hint’ instructions are properly scheduled (the compiler is generally very good at this, but …)
  - avoid data-dependent branches where possible (e.g., by using ‘select’ operation)

- **SPE fixed-point arithmetic**
  - avoid multiplication of ints (SPE does not have a 32x32 fixed-point multiply)
  - keep in mind that SPE has no fixed-point saturating arithmetic

- **SPE intrinsics**
  - use inline assembly intrinsics to control of instruction scheduling, data loads/stores, looping and branching, etc.

- **SPE issue behavior**
  - SPE contains 2 instruction pipelines with instructions pre-assigned to execute on only one of the pipelines
  - two instructions are issued every clock cycle assuming
    - there are no dependencies and operands are available
    - the even addressed instruction is a pipeline 0 instruction and the odd addressed instruction is a pipeline 1 instruction
    - the instructions are ordered pipeline 0 followed by pipeline 1
  ➔ Need to choose the instructions wisely to improve dual issue rates and reduce latencies
SPE Static Timing Analysis

- Static timing analyzer is provided in the SDK with the SPE XLC compiler
  - version for use with SPE gcc compiler also provided

- Provides a static timing analysis of compiled SPE code
  - based on issue rules, pipeline latencies, and static dependencies
  - assumes all branches not taken
  - cannot account for data-dependent behavior

- To use the static timing analysis tool:
  - set environment variable SPU_TIMING=1
  - execute ‘make SPE_code_filename.s’ (assembler output from compiler)
  - file SPE_code_filename.s.timing also generated
Typical Cell Solutions Development Questions

- What are the core algorithms of the application area and their characteristics/requirements?
  - Languages (C, C++, Fortran … POSIX sockets & threads, MPI, OpenMP, UPC)
  - Single Precision or Double Precision
  - Time to solution examples and problem scaling
  - Typical kLoC
  - Library dependencies
  - Tools/ISV dependencies

- What is the per CPU memory requirement? memory b/w requirement? I/O requirement?

- What is the characteristic ops/byte?

- What are the interconnect requirements for interprocessor communication – latency and bandwidth?

- What type of programming model does the application support, e.g. SIMD, MIMD, Embarrassingly Parallel, Collective Communication, Shared Memory …

- What is the current preferred platform for these applications and why?

- Are there any further application requirements (e.g. dynamic and resilient infrastructure, security, heterogeneous data sources, etc) that are important when deciding on system architecture?

- What are some example ISVs or Open Source implementations?

- Are there any applicable benchmarks (e.g. GUPS, SPEC)?

- How could you envision Cell as an accelerator for these algorithms?
Cell Solutions Affinity Areas

- **Digital Media**
  - Image processing
  - Video processing
  - Visualization
  - Compression / decompression
  - Encryption / Decryption
  - Digital Signal Processing
  - Digital Content Creation

- **Graphics**
  - Graphics Transformations
  - Lighting
  - Ray Tracing
  - Ray casting
  - Rich Media Mining

- **Floating Point Intensive Applications**
  - Single precision Physics
  - Single precision HPC
  - Sonar

- **Pattern Matching**
  - String manipulation (search engine)
  - Parsing, transformation, and Translation (XSLT)
  - Audio processing
  - Language translation
  - Speech recognition
  - Filtering & Pruning

- **Offload Engines**
  - TCP/IP
  - Math
  - Network Security and Intrusion

- **Biology**
  - Genomics
  - Gene Sequencing
  - Medical Imaging
  - Bioinformatics

- **Business Intelligence**

- **Financial Service**

- **Streaming Data processing Applications**

- **High-Energy Physics**

- **Astrophysics**

- **CFD**

- **Molecular Dynamics**

- **Radar Cross Section**

- **High-Resolution Atmospheric/Ocean**

- **Electromagnetics**

- **Computational Chemistry**

- **Crash Simulation**

- **Monte Carlo**

- **Low-Res Climate**

- **Seismic Migration**

- **Reservoir Simulations**

- **NVH/Structures**

- **EDA**

- **Networking**
Ideal Cell Software

- **Algorithms that fit GPU’s (Graphics Processing Units)**
  - GPU’s are being used for more than just graphics today thanks to PCI Express
  - A list of types and examples ported to GPUs can be found at: http://www.gpgpu.org/ and is partially duplicated here:

<table>
<thead>
<tr>
<th>Categories</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPGPU (182)</td>
<td></td>
</tr>
<tr>
<td>Advanced Rendering</td>
<td>20</td>
</tr>
<tr>
<td>Global Illumination</td>
<td>9</td>
</tr>
<tr>
<td>Image-Based Modeling &amp; Rendering</td>
<td>5</td>
</tr>
<tr>
<td>Audio and Signal Processing</td>
<td>3</td>
</tr>
<tr>
<td>Computational Geometry</td>
<td>10</td>
</tr>
<tr>
<td>GIS</td>
<td>1</td>
</tr>
<tr>
<td>Surfaces and Modeling</td>
<td>2</td>
</tr>
<tr>
<td>Conferences</td>
<td>3</td>
</tr>
<tr>
<td>Contests</td>
<td>1</td>
</tr>
<tr>
<td>Database</td>
<td>4</td>
</tr>
<tr>
<td>Sort &amp; Search</td>
<td>1</td>
</tr>
<tr>
<td>GPUs</td>
<td>6</td>
</tr>
<tr>
<td>High-Level Languages</td>
<td>11</td>
</tr>
<tr>
<td>Image And Volume Processing</td>
<td>29</td>
</tr>
<tr>
<td>Computer Vision</td>
<td>3</td>
</tr>
<tr>
<td>Med &amp; Bio</td>
<td>1</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>36</td>
</tr>
<tr>
<td>Books</td>
<td>5</td>
</tr>
<tr>
<td>Courses</td>
<td>11</td>
</tr>
<tr>
<td>Developer Resources</td>
<td>12</td>
</tr>
<tr>
<td>Journals</td>
<td>2</td>
</tr>
<tr>
<td>Research Groups</td>
<td>2</td>
</tr>
<tr>
<td>Press</td>
<td>1</td>
</tr>
<tr>
<td>Scientific Computing</td>
<td>40</td>
</tr>
<tr>
<td>Data Compression</td>
<td>2</td>
</tr>
<tr>
<td>Data Structures</td>
<td>1</td>
</tr>
<tr>
<td>Dynamics Simulation</td>
<td>1</td>
</tr>
<tr>
<td>Numerical Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>Site News</td>
<td>6</td>
</tr>
<tr>
<td>Stream Processing</td>
<td>1</td>
</tr>
<tr>
<td>Tools</td>
<td>8</td>
</tr>
</tbody>
</table>
Ideal Cell Software (up to 100x)

- Problem can be re-coded
- Typical code is double-buffered gather-compute-scatter
- Structured
  - Predictable non-trivial memory access pattern
  - Can build scatter-gather lists
  - Easier for memory fetch & SIMD operations
  - Data prefetch possible
  - Non branchy instruction pipeline
  - Data more tolerant, but has the same caution
- Multiple Operations on Data
  - Many operations on same data before reloading
- Easy Parallelize and SIMD
  - Little or nor collective communication required
  - No Global or Shared memory or nested loops
- Compute Intense
- Fits Streaming Model
  - Small computation kernel through which you stream a large body of data
  - Algorithms that fit Graphics Processing Units
  - GPU's are being used for more than just graphics today thanks to PCI Express
- Focus on 32b float, or <=32b integer
- Examples:
  - FFTw (best result about 100GFlops)
  - Terrain Rendering Engine
  - Volume rendering
  - Crypto codes (RSA, SHA, DES, etc. etc. etc.)
  - Media codes (MPEG 2, MPEG 4, H.264, JPEG)
Non-Ideal Software

- **Branchy data**
  - Instruction “branchiness” may be partially mitigated through different methods (e.g. calculating both sides of the branch and using select)

- **Not structured**
  - Not SIMD friendly

- **Pointer Indirection** or **multiple levels of pointer indirection** (fetching becomes hard)

- Data load granularity less than 16 bytes (will cause performance degradation)
  - DMA <128 Byte
  - SPE to local store is <16 Byte

- **Not easily parallelized**

- **Tightly coupled algorithms** requiring synchronization