Cell Broadband Engineの性能を自然に引き出すための考え方と実行環境

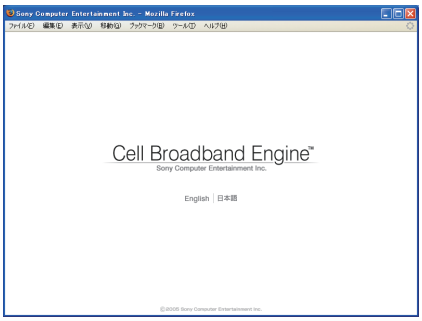
（株）ソニー・コンピュータエンタテイメント
井上敬介

目次

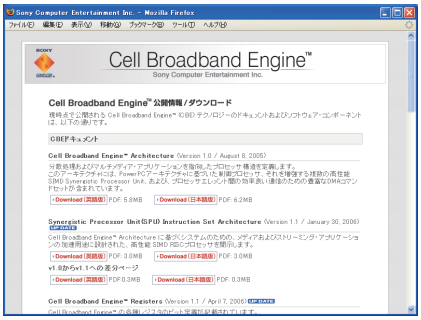
Cell Broadband Engineの概要

- 仕様は一般公開されている
- Cell Broadband Engineの特徴
- Cell Broadband Engineの最適化ポイント

http://cell.scei.co.jp/



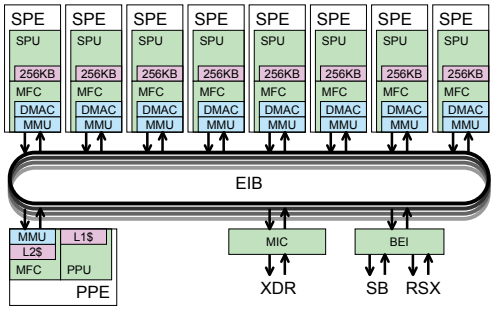
日本語ドキュメントやソフトウェア



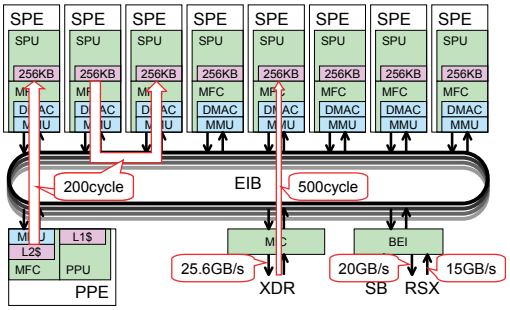
Cell Broadband Engineの概要

- 仕様は一般公開されている
- Cell Broadband Engineの特徴
- Cell Broadband Engineの最適化ポイント

Cell Broadband Engineのブロック図

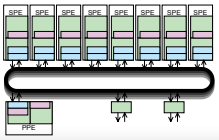


バンド幅とレイテンシ



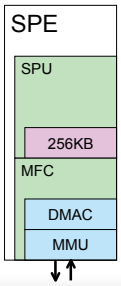
Cell Broadband Engineの並列処理的側面

- ヘテロジニアスなチップマルチプロセッサ
  - SPE視点ならシンメトリック
- キャッシュコヒーレンシ
  - PPEのキャッシュ
  - SPEのアトミックキャッシュ（同期用）
  - SPEのDMA



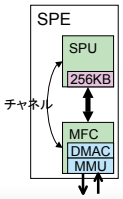
SPEの概要

- SPU - Synergistic Processing Unit
  - シンプルな構造の普通のプロセッサ
    - シンプルで静的な振る舞い
    - 動作の予測が容易
  - 128本の128ビットレジスタファイル
  - ほとんどSIMD命令
  - 256KBローカルストレージ（LS）
    - L1キャッシュ相当のスピード
    - 固定レイテンシ
    - コードとデータで共用
- MFC - Memory Flow Controller
  - DMAコントローラ
  - Memory Management Unit



SPEの並列処理的側面

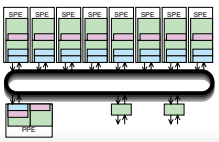
- SPUとMFCは別プロセッシングユニット?
  - メッセージパッシング
    - SPU⇔MFCはチャネルインタフェースを通して通信
      - リクエスト→ステータスリード（ブロッキング、ノンブロッキング）
  - LS⇔MFCは別パス
  - お互いに独立動作



Cellの性能を存分に生かすには？

- 当然SPEを効率よく使い切る
- あらゆるレベルで並列処理
  - Processing Element
  - XDR⇔LS転送と計算
  - LS⇔レジスタ転送と計算
  - Dual Issue
  - SIMD

ここが勝負

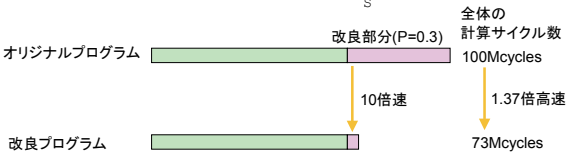


Cell Broadband Engineの概要

- 仕様は一般公開されている
- Cell Broadband Engineの特徴
- Cell Broadband Engineの最適化ポイント

アムダールの法則

- 改良部分の全体に対する割合：P
- 改良部分の性能向上率：S
- 全体の性能向上率は $\frac{1}{1-P+\frac{P}{S}}$



Cell上での最適化の基本スタンス

- CellはSPUを使用しなければ「遅いPowerPCプロセッサ」
- PPUは初期化、SPU起動、システム制御等「のみ」を行うのが理想
- SPUはいわゆるコプロセッサではない（EEのVUとは異なる）
- PPUは1個しかないが、SPUは複数使える（台数効果）

Cell上での最適化では、
あらゆる手段を講じてSPUを活用すべき

High Level Optimization (Algorithmic)

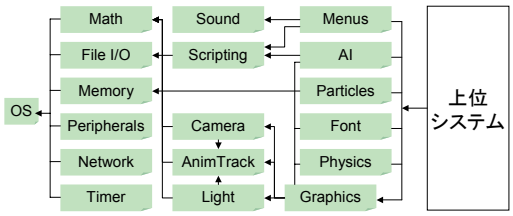
Cellに向いたアルゴリズム／リファレンスコードの選択

- 命令数削減
  - 演算量／データ転送量が少ないアルゴリズムの適用
  - 効果的なSIMDアルゴリズムの適用
- メモリバンド幅低減
  - Local Storageの活用（LS内データ再利用）
  - 効果的なDMAのためのデータ構造
- 複数SPUでの台数効果
  - 負荷分散
  - 同期オーバヘッド削減
  - SPU間のデータ転送量削減

Low Level Optimization (Implementation)

- サイクル数削減
  - DMAレーテンシ隠蔽
    （DMAと演算のオーバラップ）
  - 分岐ペナルティ削減
  - Dependency Stall削減
  - Dual Issue Rate向上

しかし、ゲームは様々な処理の集合体



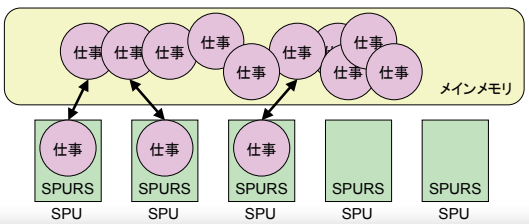
どうやってSPUに乗せて動かす？

目次

- Cell Broadband Engineの概要
  - 仕様は一般公開されている
  - Cellの特徴
  - Cellの最適化ポイント
- SPU Runtime System
  - SPURSの概要
  - SPURSタスク（並列実行モデル）
  - SPURSジョブ（並列実行モデル）
  - SPURSカーネル（インテグレーション）

SPURSとは？

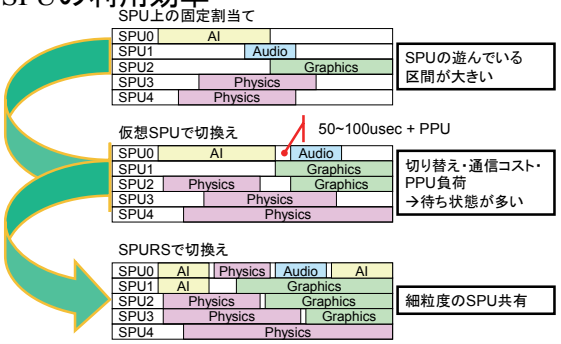
- SPURS = SPU Runtime System
- SPU自身によるSPUの仕事切替機構



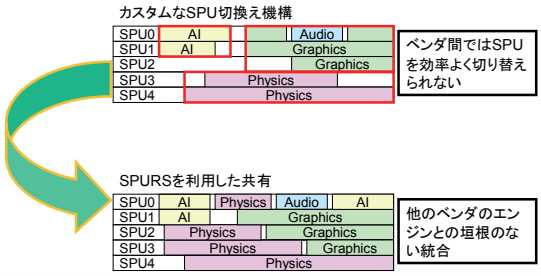
なぜSPU中心の自律的なモデルか？

- Cellに自然なモデルを提供し、Cellの性能を自然に引き出す
- PPUから逐一指示を出していたのでは
  - SPUから使いづらいデータ構造になりがち
  - SPUをコプロセッサとして使いがち
    - 密結合でないコプロセッサは通信コストで負ける
  - SPUへ関数オフロードしがち
    - 判断基準がSPU向きの処理かどうかではアムダールの法則により効果薄

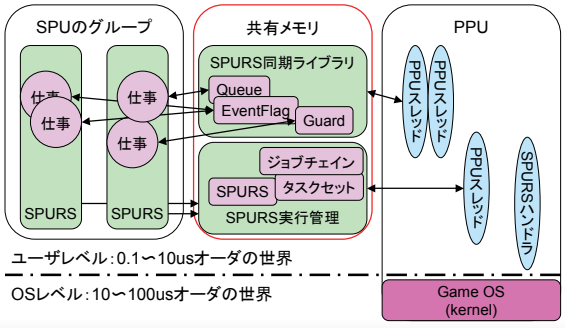
なぜSPURSが必要か？ SPUの利用効率



なぜSPURSが必要か？ 他のベンダのEngineとの統合



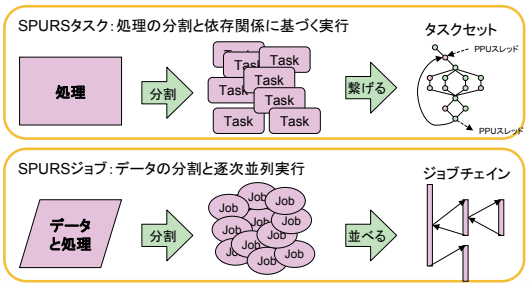
SPURSの概観



ロックフリー実装

- OSのmutexはSPUの自律実行を止める
  - SPUは特権モードが無い
  - SPUのOS呼出しはリモートPPUコール
    - 同期PPUコールはSPUを止める
    - PPUコールはPPUも止める
- ユーザレベル排他（mutex）は難しい
  - PPUスレッドのスケジュールはSPU側と無関係
  - PPUスレッドはプリエンプションされる
    - PPUがロックを取ったままどこかに行ってしまうと、SPUは待ちぼうけになる

2つの並列プログラミングモデル



SPURSタスクとSPURSジョブのどちらを使うか

SPURSタスク
- コンベンショナルなマルチSPU・マルチタスクモデル
- コンベンショナルなプログラミングスタイル
- ロックフリー実装の軽量同期ライブラリ群

自分で制御したい人向け

自動で制御してもらいたい人向け

SPURSジョブ
- データ指向のマルチSPU・並列データ処理モデル
- 入力データを処理して出力データを生成するプログラミングスタイル
- 入力DMA、実行、出力DMAの自動オーバーラップ（パイプライン実行）

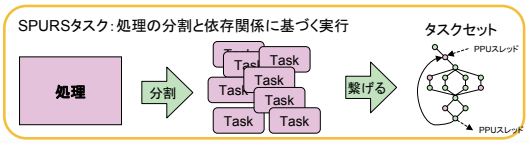
50k～100kサイクルの処理に最適

SPU Runtime System

- SPURSの概要
- SPURSタスク（並列実行モデル）
  - タスクセット
  - タスクの状態遷移
  - タスクの選択と実行
  - タスク切り替えのシーケンス
- SPURSジョブ（並列実行モデル）
- SPURSカーネル（インテグレーション）

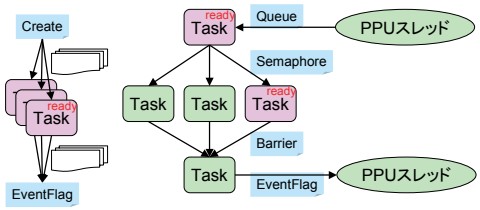
SPURSタスク実行モデルの概観

- コンベンショナルなマルチSPU・マルチタスクモデル
- コンベンショナルなプログラミングスタイル
- ロックフリー実装の軽量同期ライブラリ群

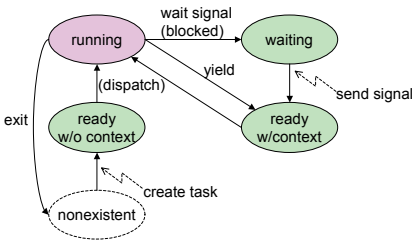


タスクセット

- タスク操作や同期ライブラリを使用した事により依存関係で結合されたタスク群

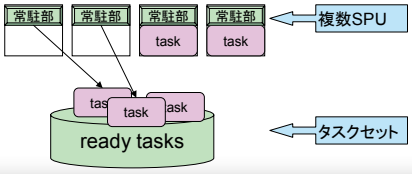


タスクの状態遷移図

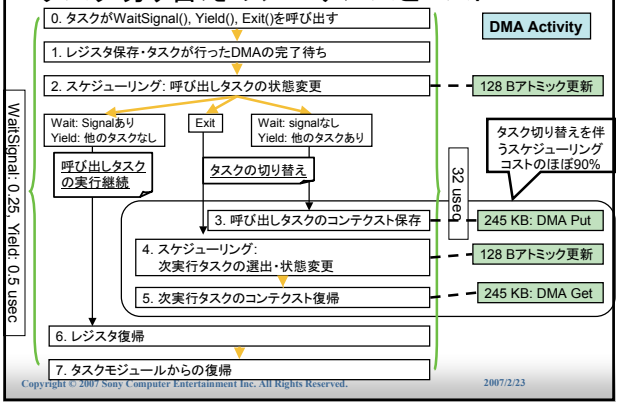


タスクの選択と実行

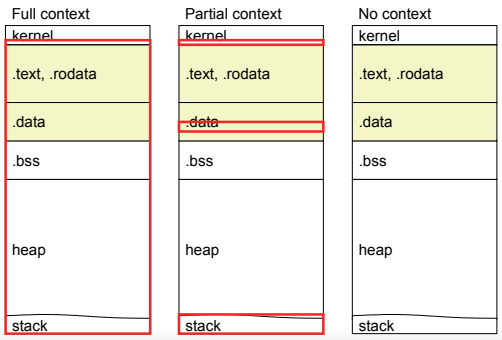
- タスクセットを複数SPUで共有
- 各SPUで独立してタスクを選択して実行
  - ロックフリー実装でアトミックにタスクを選択
  - FIFOでないが公平なスケジューリング



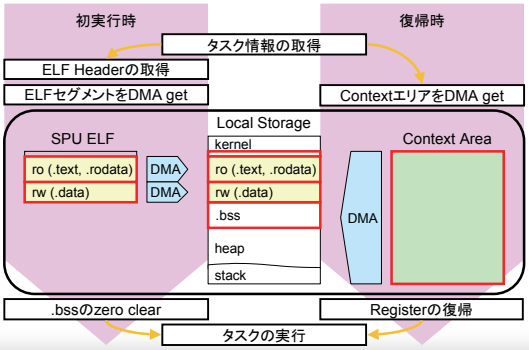
タスク切り替えのシーケンスとコスト



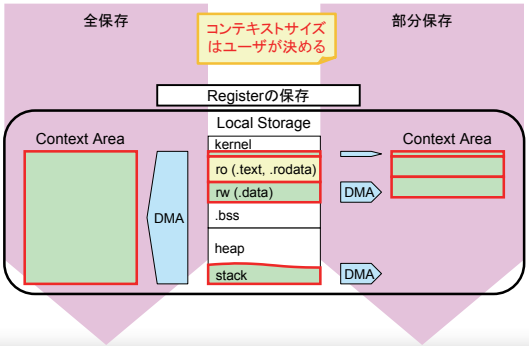
タスクのLSコンテキストサイズは任意



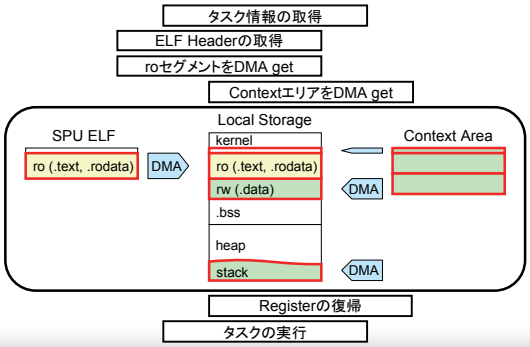
タスクコンテキストの生成・復帰



タスクコンテキストの保存



タスクコンテキストの部分復帰



SPURSタスク実行モデルのまとめ

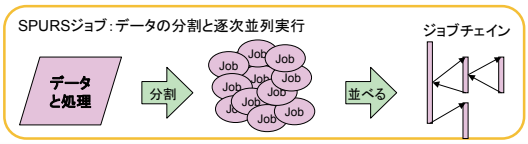
- ノンプリエンプティブ、マルチSPUマルチタスク
- 固定メモリレイアウト
  - ユーザメモリエリアはLSの95%以上
- ロックフリーデザイン
  - PPUスレッドもSPURSタスクと協調動作可能
  - 同期操作のPPUスレッドがプリエンプトされてもSPU側に影響なし
- 軽い動作の同期・通信ライブラリ
  - PPUスレッドをブロックする以外にシステムコールは呼ばない

SPU Runtime System

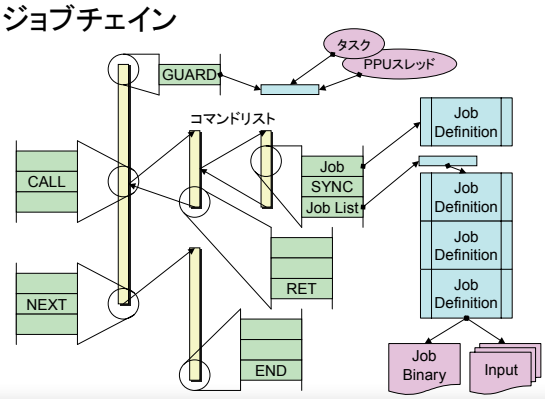
- SPURSの概要
- SPURSタスク（並列実行モデル）
- SPURSジョブ（並列実行モデル）
  - ジョブチェイン
  - コマンド処理
  - ジョブのパイプライン実行
- SPURSカーネル（インテグレーション）

SPURSジョブ実行モデルの概観

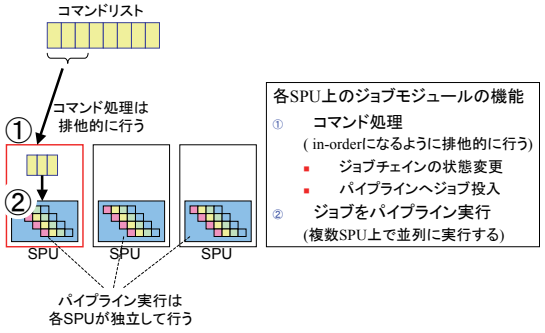
- データ指向のマルチSPU・並列データ処理モデル
- 入力データを処理して出力データを生成するプログラミングスタイル
- 入力DMA、実行、出力DMAの自動オーバーラップ（パイプライン実行）
- ロックフリー実装の同期ライブラリ群



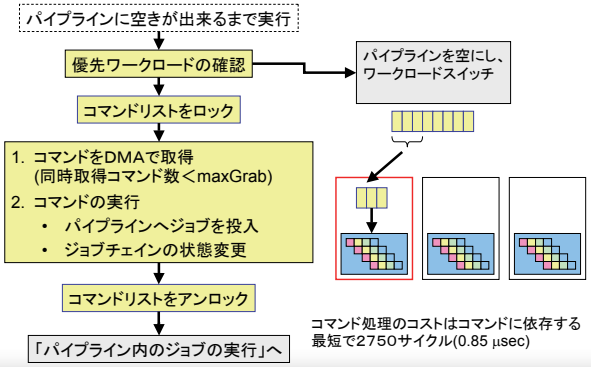
ジョブチェイン



ジョブチェインの動作

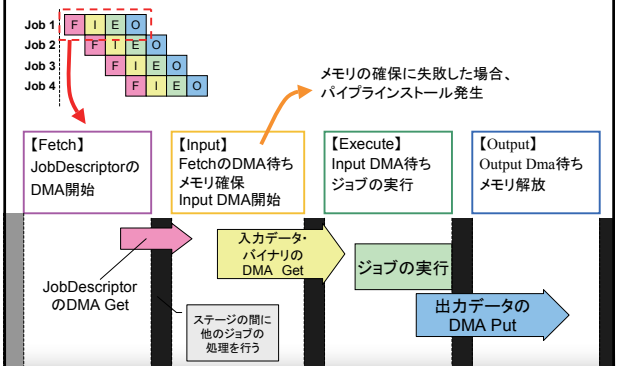


コマンド処理時の動作

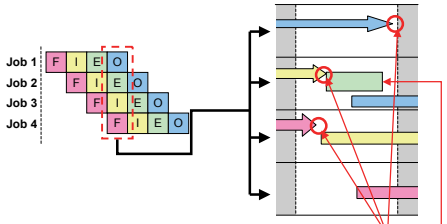


パイプラインの各ステージの処理

[libgcm概要14章]



1ステージの実行時間



1ステージの実行時間
＝前ステージのDMA待ち時間
＋ジョブの実行時間
＋オペレーションコスト ← 1900cycle (0.6 μsec)

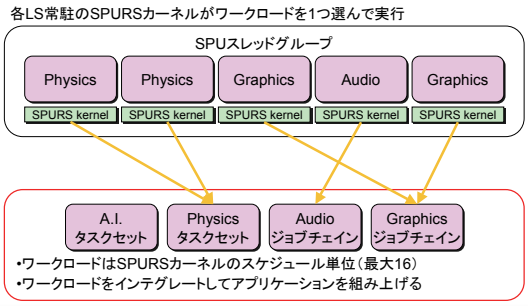
SPURSジョブ実行モデルのまとめ

- ジョブチェインを複数SPUで実行
- ジョブはSPU上で4ステージパイプライン実行
  - ジョブとDMAも並列実行
- 柔軟なメモリ管理（mallocは未使用）
  - ジョブはほぼ全メモリの使用も可能
    - パイプラインストールが発生
  - バイナリやroデータをキャッシュ可能
- 優先ジョブコマンド（Urgent Job）
- ジョブチェイン、ジョブリストは再利用可能

SPU Runtime System

- SPURSの概要
- SPURSタスク（並列実行モデル）
- SPURSジョブ（並列実行モデル）
- SPURSカーネル（インテグレーション）

SPURSカーネルの概観

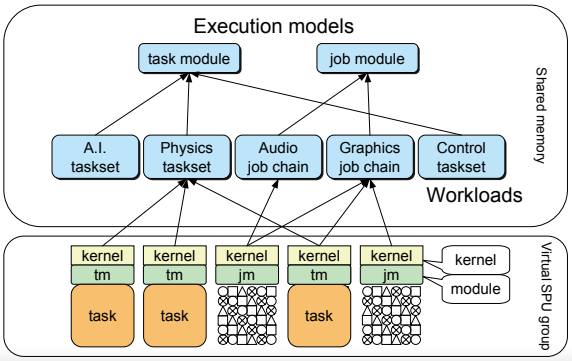


スケジューリングの特徴

- Non-preemptive
  - 低コストで効率よくSPUを譲り合う
    - 誰かがSPUを要求（ReadyCount）
    - 優先ワークロードの確認（cellSpursModulePoll）
    - SPUを譲る（cellSpursModuleExit）
- 優先度ベース
  - SPU毎に設定
    - インテグレータがSPUの割振りをしやすい

コンテキスト破棄

SPURSの全体像



SPURSのまとめ

- SPURSは2レイヤ構成
  - 2KBのワークロードスケジュール層
  - 実行モデル（ポリシーモジュール）層
- 効率の良い実行
- 協調的なSPUの共有
- インテグレータに嬉しい見通しの良い動作
  - No virtualization
  - Non-preemptive
  - Static behavior of scheduling