AMD GPU プログラミング

N.Nakasato

GPUのプログラミング(1)

- 2000年頃以降
  - programable shaderが搭載されるようになる
    - HLSL, GLSL, Cg等のシェーディング言語で、shaderをプログラム可能になった
      - ただし、制限が多かった
  - General Purpose GPU programming
    - GPUでの汎用計算(GPGPU)
- 2006年以降
  - programable shaderでの様々な制限が取り払われる(Nvidia G80, AMD R600)
    - Nvidia社よりCUDA(Compute Unified Device Architecture)を公開

GPUのプログラミング(2)

- 2007年
  - CUDAを利用した様々なアプリケーション
  - AMD/ATi社もR600でG80と同様な進化をするが、CUDAのようなプログラミング環境を提供せず
- 2008年
  - AMD社は、R600を改良したR700アーキテクチャのGPUを発売
  - CAL/Brook+を提供開始

CALとは(1)

- Compute Abstraction Layer
- AMD社のGPUをプログラムするためのソフトウエア環境

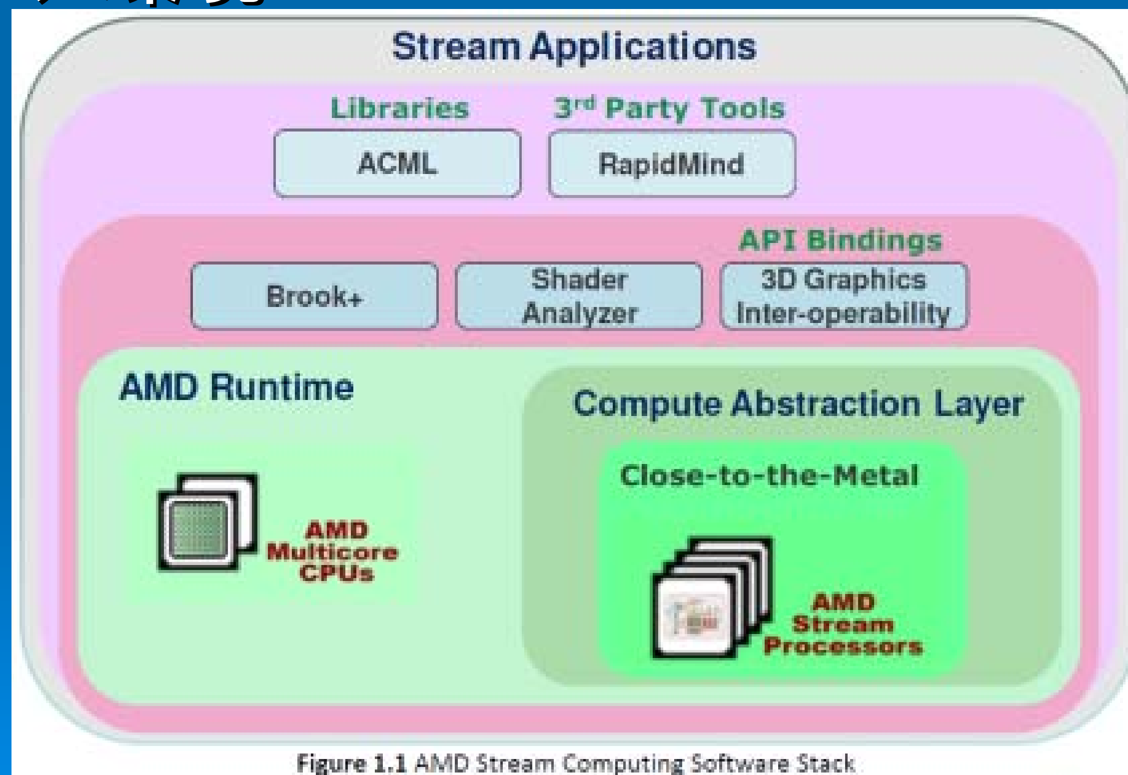


Figure 1.1 AMD Stream Computing Software Stack

AMD Compute Abstraction Layer Programming Guideより

CAL(2)

- CALできることとは
  - GPUボードの管理
  - リソース(メモリ)の管理
  - GPU用コードの生成
  - kernelのロードと実行
  - 複数ボードのサポート
  - 以上に対応したAPIがそろっている
- CALを使うことでGPUを様々な計算に利用できるようになる。

CALのアーキテクチャ(1)

- CPUからGPUを操作する
  - アプリケーションは、CPUで実行されるコードと、GPUで実行されるkernelからなる
  - APIはCPUで実行される

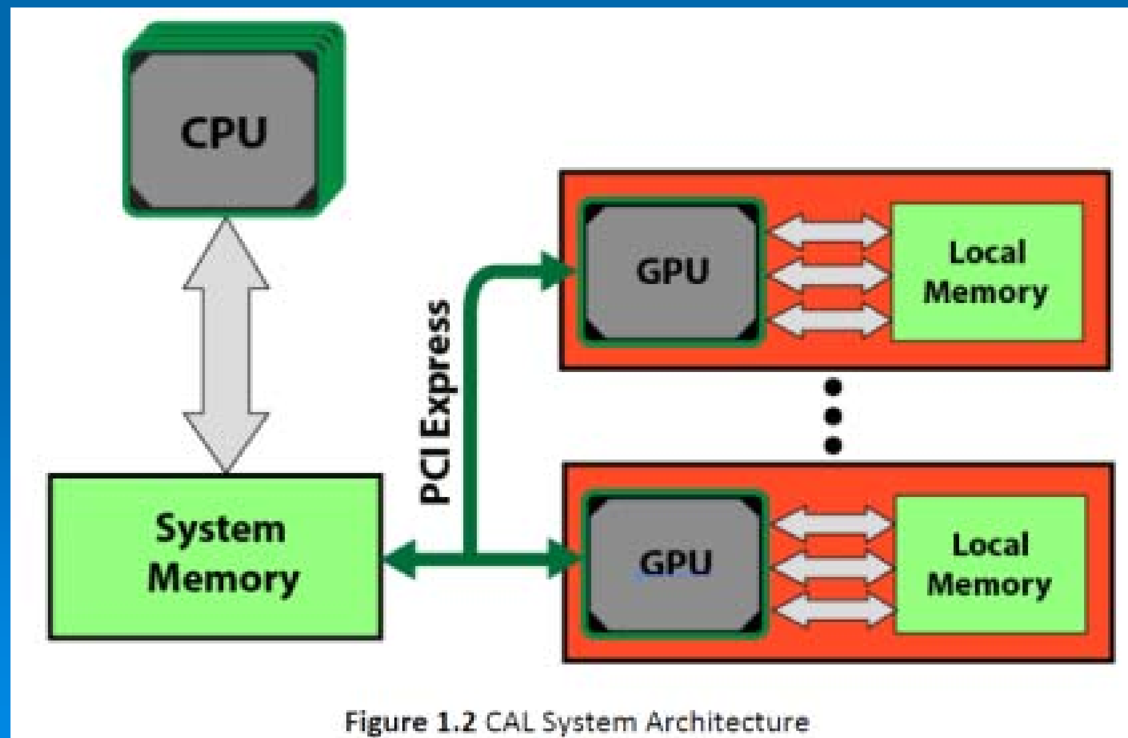


Figure 1.2 CAL System Architecture

CALのアーキテクチャ(2)

- CPUとGPUはそれぞれ自分のメモリをもつ
  - CALではGPU側からの視点でメモリを扱う
    - CPUのメモリ：system/remote memory
    - GPUのメモリ：GPU/local memory
- CPUはsystemメモリのみアクセス可能
- GPUはどちらもアクセス可能
  - GPUメモリは高速な内部バスを通して
  - SystemメモリはPCI Expressを通して

CALデバイス(1)

- CALでのGPUボードはSIMDプロセッサの集団として抽象化されている
- このSIMDプロセッサの集団が、ロードされたkernelを実行する
- kernelは、input resource(メモリと同義)からデータを読み、計算を実行し、結果をoutput resourceに書き出す
  - どちらも複数のinput/outputを指定可能
  - これらのresourceはそれぞれ長方形のdomain
    - ２次元配列のメモリと思えばよい

CALデバイス(1)

- Resourcesはlocalにもremoteにもおける
  - GPUは自分のメモリ(local)とCPUのメモリ(remote)の両方にアクセス可能
  - それぞれAPIがあるので後で説明

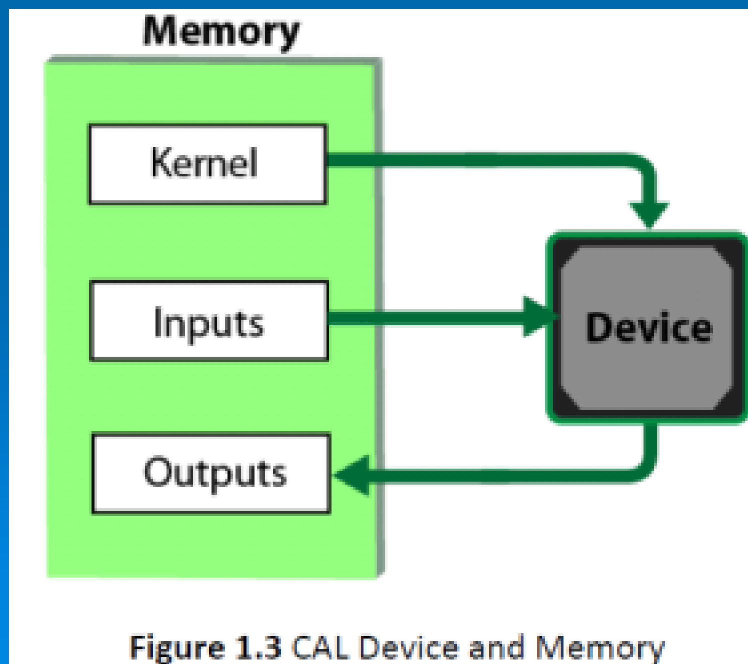


Figure 1.3 CAL Device and Memory

GPUのアーキテクチャ(1)

- AMD (ATi)社のGPUのおおざっぱな歴史
  - R600以前は、pixelシェーダーとvertexシェーダーが分離されていた。これらはプログラム可能だが、GPUでの汎用計算には不向きだった。
  - R600アーキテクチャから、unifiedシェーダーとしてSIMD arrayが実装された
    - 320個の積和演算ユニット: ～475 GFLOPS
    - Very Long Instruction Word (VLIW)プロセッサ
  - R700アーキテクチャでは、R600アーキテクチャにGPGPU向けの改良を施した
    - 800個の演算ユニット: ~1.0 – 1.2 GFLOPS
    - 倍精度演算の高速化

GPUのアーキテクチャ(2)

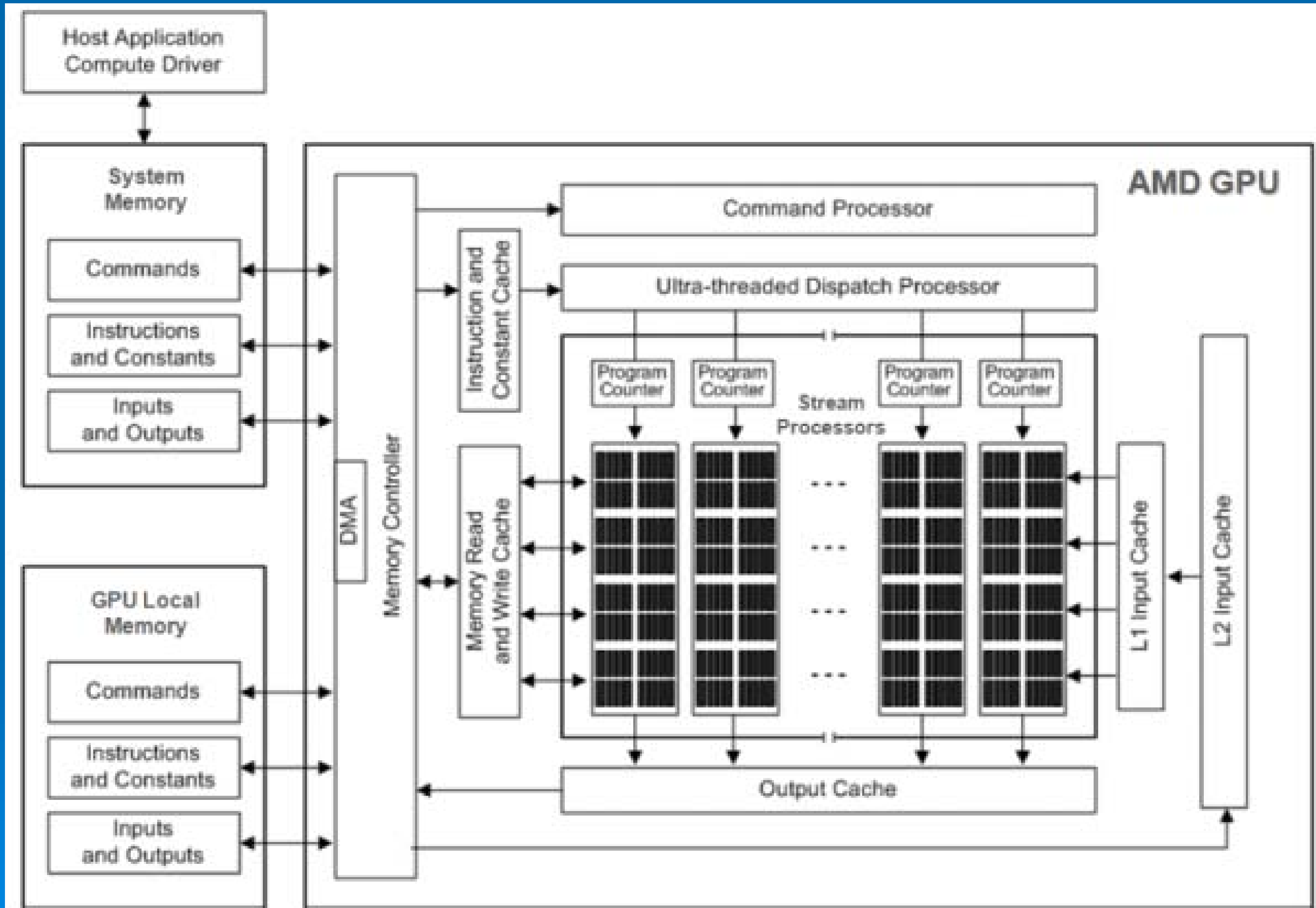


Figure 1.4 AMD GPU Architecture

GPUのアーキテクチャ(3)

- Command processor (CP)
  - CPUがロードしたkernelの命令を発効する。計算が終了したCPUに知らせる。
- Stream processor array (SP)
  - それぞれ独立したSIMD演算ユニット。並列に動作する。
- Memory controller (MC)
  - Localメモリに直接アクセス可能。Systemメモリの一部もアクセス可能。DMA転送もできる
  - Memory controllerとprocessor arrayの間には、データ・命令用の複数のcacheがある

GPUのアーキテクチャ(4)

- アプリケーションの実行時には
  - Kernelはホスト(CPU)により制御される。実行前ホストは、CPにdata domain等のデータを送る。
  - その上で、CPは個々のSPでのkernelの実行を開始する。この時に、CPからそれぞれのSPには、SPを特定するためのindex pair (x, y)が送られる。
  - このpairを使って、個々のSPは、自分の担当するinput/outputデータをロードしたり、セーブする。
  - SPは、MCを通してlocalメモリからデータを自分のレジスタ(GPR)にロードしつつ、プログラムを実行する。
    - 命令はCPによりbroadcastされる(おそらく)

CALでの実行モデルまとめ(1)

- アプリケーションは、ホスト(CPU)とGPUからなる。ホストがGPUを制御する。
  - CellでのPowerPCとSPEに対応
- ホストは、計算データをGPUのメモリに送る
- ホストは、kernel （GPUで実行されるプログラム)をGPUに送る。
- GPU内部ではCPが、kernelの実行を複数のSPに割りあてる。
  - この時、それぞれのSPには、自分の「番号」(index pair)が割りあてられる

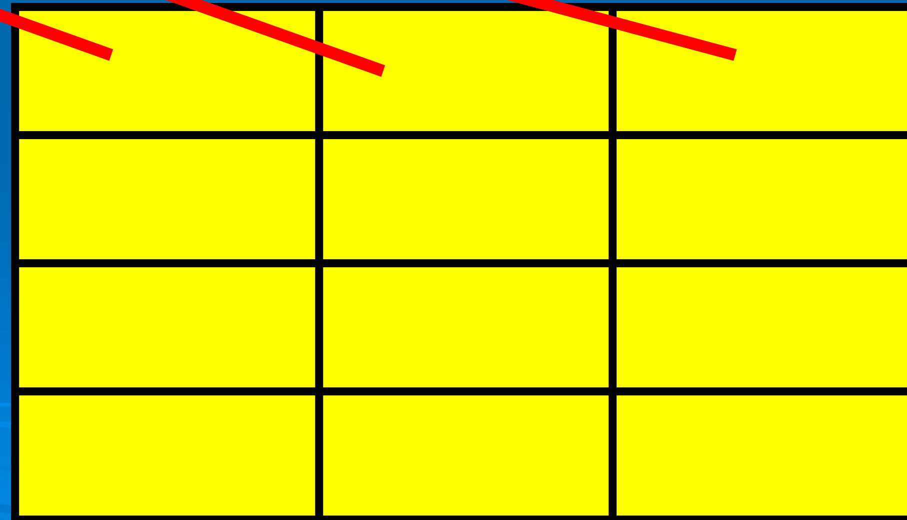
CALでの実行モデルまとめ(2)

- 各SPでは、同じプログラム(kernel)が並列実行される。
  - 複数SPでのSIMD実行
  - 全く同じ処理をしては並列実行の意味がないので、SPはメモリから異なるデータをloadし、計算をする
  - 計算結果も、所定のメモリ領域にsaveする
- SPでの処理が終わったら、CPはホストにそれを通知する
- ホストは結果をGPUのメモリから回収

CALでの実行モデルまとめ(3)

- 個々のSPはindex pairにしたがって、所定のメモリ領域からデータをload/save

SP (0,0)	SP (0,1)	SP (0,2)
SP (1,0)	SP (1,1)	SP (1,2)
SP (2,0)	SP (2,1)	SP (2,2)
SP (3,0)	SP (3,1)	SP (3,2)



GPUのメモリ

CAL API

- 前提
  - CAL APIには“cal”というprefixがつく
  - CAL APIで利用する変数や構造体には“CAL”というprefixがつく。

CAL API : 初期化(1)

- 一番最初に呼ぶ関数:
  - calInit()
- デバイス(GPUボード)の数を得る:
  - calDeviceGetCount(CALuint *)
- デバイスをopen:
  - calDeviceOpen(CALdevice *, int)
  - 引数は、デバイスハンドルとデバイスの番号(複数ボードの場合)

CAL API : 初期化(2)

- デバイスの種類を問い合わせ:
  - calDeviceGetAttribs(CALdeviceattribs *, int);
  - 引数は、デバイス用構造体とデバイス番号
  - この構造体には、デバイスのハードウエアに関する情報がはいっており、あとでkernelのコンパイル・リンク時に必要

```
typedef struct CALdeviceattribsRec {
    CALuint    struct_size;        /**< Client filled out size of CALdeviceattribs struct */
    CALtarget  target;             /**< Asic identifier */
    CALuint    localRAM;           /**< Amount of local GPU RAM in megabytes */
    CALuint    uncachedRemoteRAM;  /**< Amount of uncached remote GPU memory in megabytes */
    CALuint    cachedRemoteRAM;    /**< Amount of cached remote GPU memory in megabytes */
    CALuint    engineClock;        /**< GPU device clock rate in megahertz */
    CALuint    memoryClock;        /**< GPU memory clock rate in megahertz */
    CALuint    wavefrontSize;      /**< Wavefront size */
    CALuint    numberOfSIMD;       /**< Number of SIMDs */
    CALboolean doublePrecision;    /**< double precision supported */
    …省略…
    CALboolean memExport;          /**< memexport supported */
} CALdeviceattribs;
```

CAL API：初期化コード

```
CALuint numDevices = 0;
CALdevice device = 0;

calInit();
calDeviceGetCount(&numDevices);
calDeviceOpen(&device, 0);

CALdeviceattribs attribs;
attribs.struct_size = sizeof(CALdeviceattribs);
calDeviceGetAttribs(&attribs, 0);
```

以上は決まり文句なので、毎回同じ

CAL API :kernel関連(1)

- CALでは、kernelをILとよばれる抽象化されたアセンブラで記述する。
  - ハードウエアの違いをここで吸収
  - 高級言語 → IL → GPU用の機械語
  - ILの変わりにHLSL等も利用可能
- ILはテキストなので、プログラム実行時に、デバイスに合わせてcompile/linkする。
  - ILをcompileしてobjectを生成
  - そのオブジェクトをlink処理

CAL API :kernel関連(2)

- CALobject : kernelのリンク前のobject
- CALimage : kernelの実行image
  - CALimageを直接読み込むAPIもある

```
CALobject obj = NULL;
CALimage image = NULL;

CALlanguage lang = CAL_LANGUAGE_IL; // 入力としてILを指定
char program[] = "….”; // ILのテキストで初期化

// ILからobjectの生成。最後の引数でGPUのアーキテクチャを指定
calclCompile(&obj, lang, program, attribs.target);
                                        // ↑先に取得したCALdeviceattribs

// objectをlinkして実行imageの生成
calclLink(&image, &obj, 1);
```

CAL API :コンテキスト(1)

- kernelをGPUで実行するためには、CALコンテキストを作成する必要がある
  - CALコンテキストには、デバイスの状態の情報が全て含まれている
    - メモリ割りあて状況
    - kernel image
    - kernelの実行時情報など
  - 一つのデバイスに、複数のコンテキストを割りあてることができる
  - 作成したコンテキストに、kernel imageを結びつけることで、kernelの実行が可能となる

CAL API :コンテキスト(2)

- コンテキストを作成:
  - calCtxCreate(CALcontext *, CALdevice)
- コンテキストを削除:
  - calCtxDestroy(CALcontext)
- モジュールの生成
  - calModuleLoad(CALmodule *p, CALcontext, CALimage)
  - コンテキストとkernel image結びつけたものがモジュール

```
CALcontext ctx;
calCtxCreate(&ctx, device);

CALmodule module;
calModuleLoad(&module, ctx, image);
```

CAL API :メモリ(1)

- CALではメモリに2種類ある
  - Remote Memory (ホストのメモリ)
  - Local Memory (GPUのメモリ)

- どちらも以下のステップで利用
  - メモリ割りあて
  - アクセスのためのポインタ取得
  - コンテクストごとのメモリハンドル取得
  - メモリハンドルをkernelの変数と束縛

CAL API :メモリ(2)

- メモリ割りあて
  - calResAllocLocal2D(CALresource *, CALdevice, CALuint, CALuint, CALformat, CALuint);
  - 2次元のローカルメモリを割りあてる
  - 引数は:リソースのハンドル、デバイス、メモリ領域の幅、高さ、メモリフォーマット、flag(未使用)
  - メモリフォーマット
    - CAL_FORMAT_FLOAT_1 : 1要素float(4 byte)
    - CAL_FORMAT_FLOAT_4 : 1要素4float(16 byte)

CAL API :メモリ(3)

- メモリ割りあて続き
  - 幅と高さは要素の数で指定
  - 例　幅と高さを8,8とした時のメモリ量
    - CAL_FORMAT_FLOAT_1なら: 8*8*4 = 256 byte
    - CAL_FORMAT_FLOAT_4なら: 8*8*16 = 1024 byte
  - 1次元メモリ割り当ては
    - calResAllocLocal1D()
  - Remoteメモリも同様に
    - calResAllocRemote2D()
    - calResAllocRemote1D()

CAL API :メモリ(4)

- ポインタ取得
  - calResMap(CALvoid **, CALuint, CALresource, CALuint)
  - 引数は:メモリ領域へのポインタ、pitch, メモリ領域のハンドル、flags(未使用)
  - pitchは、メモリ領域の幅方向の要素数
  - メモリ領域にアクセスするためのopen()関数のようなもの
  - 利用が終わったらcalResUnmap(CALresource)でclose()相当の処理をする

CAL API :メモリ(5)

- GPUメモリに書き込む例

```
CALresource inputRes;
float *fdata;
CALuint pitch;

// 2次元メモリ(256x256)を1要素1単精度変数で確保
calResAllocLocal2D(&inputRes, device, 256, 256, CAL_FORMAT_FLOAT_1, 0);

// メモリのポインタをfdataに取得
calResMap((CALvoid**)&fdata, &pitch, inputRes, 0);

// 256x256のメモリ領域を0で初期化
for(int i = 0; i < 256; ++i) {
    float* tmp = &fdata[i * pitch]; // pitchを使って行ごとのポインタ初期化
    for(int j = 0; j < 256; ++j) {
        tmp[j] = 0.0;
    }
}
// inputResへのアクセス終了
calResUnmap(inputRes);
```

CAL API :メモリ(6)

- 読み出しも同様(FLOAT_4の場合)

```
CALresource outputRes;
float *fdata;
CALuint pitch;

// 2次元メモリ(256x256)を1要素4単精度変数で確保
calResAllocLocal2D(&outputRes, device, 256, 256, CAL_FORMAT_FLOAT_4, 0);

// メモリのポインタをfdataに取得
calResMap((CALvoid**)&fdata, &pitch, outputRes, 0);
for(int i = 0; i < 256; ++i) {
    float* tmp = &fdata[i * pitch * 4]; // 行ごとのポインタ初期化。 *4 に注意
    for(int j = 0; j < 256; ++j) {
        // 1要素(4変数)を表示
        printf("%f %f %f %f¥n",
                    tmp[j*4],tmp[j*4+1],tmp[j*4+2],tmp[j*4+3]);
    }
}
// アクセス終了
calResUnmap(outputRes);
```

CAL API :メモリ(7)

- 割りあてたメモリを利用するには
  - メモリをコンテキストに束縛し
  - さらにそのコンテキストのkernel内の変数と束縛する必要がある

- コンテキストに束縛
  - calCtxGetMem(CALmem *, CALcontext, CALresource);
  - CALmemが束縛したコンテキストでのメモリハンドル

CAL API :メモリ(8)

- kernel内の変数のハンドル取得
  - calModuleGetName(CALname *, CALcontext, CALmodule, CALchar *)
  - 引数:CALnameが変数のハンドル, コンテキスト、module、取得した変数名
- 変数のハンドルとメモリを束縛
  - calCtxSetMem(CALcontext, CALname, CALmem)
  - 引数:コンテキスト, 変数のハンドル、メモリのハンドル

CAL API :メモリ(9)

- メモリを使うには
  1. メモリ領域確保
  2. メモリ領域を現コンテキストのkernel変数に束縛
  3. メモリにデータを書き込む・読み出す
- 以上の処理をおこなう必要がある
  - ただし、1と2の処理は一つのコンテキストに対して、一度おこなえばよい
- データを繰り返しupdateするようなアプリケーションでは、3の処理のみを繰り返せばよい

CAL API : kernel実行(1)

- kernelの実行を開始
  - calCtxRunProgram(CALevent *, CALcontext, CALfunc, CALdomain)
  - 引数:CALeventはこのkernel実行に付随したハンドル, コンテキスト, kernelのmain関数のハンドル, kernelが実行される矩形領域
  - CALfuncはcalModuleGetEntry(CALfunc *, CALcontext, CALmodule, CALchar *)で取得する
  - CALdomainは4要素の整数配列で矩形領域を指定する
    - {0,0,100,100} : 100x100の領域

CAL API : kernel実行(2)

- kernelの終了を待つ
  - calCtxIsEventDone(CALcontext, CALevent)

```
CALfunc func;
CALevent e;
CALdomain domain = {0,0,128,128}

// kernelのmain関数へのハンドルを取得
calModuleGetEntry(&func, ctx, module, "main");

// kernelの実行開始
calCtxRunProgram(&e, ctx, func, &domain);

// 終了待ち
while (calCtxIsEventDone(ctx, e) == CAL_RESULT_PENDING);

//以下結果読み出し等の処理
```

CAL API：後処理

- モジュールのunload
  - calModuleUnload()
- Imageとobjectの領域解放
  - calFreeImage(), calFreeObject()
- コンテキストに束縛したメモリを解放
  - calCtxReleaseMem()
- メモリ領域を解放
  - calResFree()
- コンテキストの解放
  - calCtxDestroy()
- デバイスをclose
  - calDeviceClose()
- CALの利用を終了
  - calShutdown()

CALのまとめ

- 基本的な処理の流れ
  1. GPUボードの確保と初期化
  2. kernelプログラムのcomiple/link
  3. メモリ領域の確保
  4. メモリやmain関数へのハンドル初期化・設定
  5. 入力データのsetup
  6. kernelの実行
  7. 結果の回収
  8. 後処理

- 5と6を繰り返し実行。
  - その他は一度おこなえばよい。

ILの基本

- IL : AMD Intermdediate Language
  - GPUをプログラムするためのアセンブリ言語
  - GPUのアーキテクチャからは独立
    - コンパイラによりILから個々のGPU用の機械語を生成する。
- ILの構造
  - 変数に型がない。命令で変数の型が決まる
    - IMUL : 整数乗算
    - MUL : 浮動小数点乗算(単精度)など
  - プログラムは、演算文(四則演算といくつかの数学関数、比較命令)と制御構造文、そして宣言文からなる

ILの命令フォーマット

- 完全な命令フォーマット

An instruction in IL Text syntax has the following general format:

```
<instr>[_<ctrl>][_<ctrl(val)>] [<dst>[_<mod>][.<write-mask>]]
   [, <src>[_<mod>][.<swizzle-mask>]]...
```

- はしょって書くと
  - 2 operands : instr dst src1
    - 例 : mov r1, r2 (r2の値をr1にコピーする)
  - 3 operands : instr dst src1 src2
    - 例 : add r1, r2, r3 (r1 = r2+r3)

命令の付加語(1)

- Control Specifiler : <instr>[_ctrl(val)]
  - 命令に付加する。これをつけることで、命令の動作を変えることができる場合がある。
  - 例 :
- Destination Modifier : <instr>[_mod]
  - 命令の実行結果を、dstに保存する前にする操作を指定できる

Table 3-1. Destination Modifiers

Modifier	Description	Example
_x2 _x4 _x8 _d2 _d4 _d8	Shift scale modifiers	add_x2 r0, r1, r2
_sat	Saturate or clamp result to [0,1]	add_sat r0, r1, r2 add_x2_sat r0, r1, r2

命令の付加語(2)

- Write mask : reg.xywz
  - レジスタはベクトルレジスタ
    - 大きさは128 bitであり、単精度変数なら4要素
  - それぞれの要素を{xyzw}で指定できる
    - reg.{x|_|0|1}.{y|_|0|1}.{w|_|0|1}.{z|_|0|1}
    - "_" : その要素には書き込まない
    - "0" : その要素を0にする
    - "1" : その要素を1にする
    - 省略すると"_"と同じとなる

命令の付加語(3)

- Write maskの使用例
  - mov r0.x, r1 (mov r0.x___, r1)
    - r1.xをr0.xにコピー。r0の他の要素は変化しない
  - mov r0.z, r1 (mov r0._z_, r1)
    - r1.zをr0.zにコピー。r0の他の要素は変化しない
  - mov r0.x_zw, r1
    - r1.x, r1.z, r1.wをr0.x, r0.z, r0.wにコピー。
    - r0.yは変化しない
  - mov r0._1_w, r1
    - r1.wをr0.wにコピーして、r0.yを1にする
  - mov r0.11, r1 (mov r0.11__, r1)
    - r1.xとr1.yを1にする

命令の付加語(4)

- Source Modifier
  - src operandを操作する付加語

Table 3-2. Source Modifiers

Modifier	Description	Example
`_invert`	Invert components (1- x)	`add r0, r1_invert, r2`
`_bias`	Elements are biased (x – 0.5)	`add r0, r1, r2_bias`
`_x2`	Multiply elements by 2.0	`add r0, r1, r2_x2`
`_bx2`	Signed scaling. Combined bias and x2 modifiers.	`add r0, r1, r2_bx2`
`_sign`	Signs Elements Elements less then 0 become -1 Elements equal to 0 become 0 Elements greater then 0 become 1	`mov r0, r1_sign`
`_divcomp(`*type*`)`	Performs division based on *type*. *type*: y, z, w, unknown	`texld_stage(0) r0, vT0_divcomp(y)`
`_abs`	Takes the absolute value of elements.	`mov r0, r1_abs`
`_neg(`*comp*`)`	Provides per element negate	`mov r0, r1_neg(xw)`

命令の付加語(5)

- Swizzle Mask
  - Wirte maskと同様にsrc operandにもmaskをつけることができる
  - 要素間の入れ替えなどに利用する
  - “reg.1234”の”1234”のそれぞれに{x|y|z|w|0|1}を指定することが可能
  - Write maskとは異なり、swizzle maskは場所が固定されているので注意。”1”の場所は必ずx要素になる

命令の付加語(6)

- Swizzle Maskの例
  - mov r0, r1.yx (mov r0, r1.yxzw)
    - r1.yをr0.xに、r1.xをr0.yに、r1.zをr0.zに、r1.wをr0.wにコピー(r1のxとyを入れ替えてr0にコピー)
  - mov r0, r1.wzyx
    - r1のxyzwをwzyxに入れ替えてr0にコピー
  - mov r0, r1.xxyy
    - r1.xをr0.xとr0.yに、r1.yをr0.zとr0.wにコピー
  - mov r0, r1.xyz0 (mov r0.xyz0, r1)
    - r1.xをr0.x, r1.yをr0.yに、r1.zをr0.zにコピーして、r0.wを0にする

レジスタ(1)

- 通常のレジスタ(general purpose register:GRP)はr0-r127
- c#はconstant register
- i#はimmediate register(即値)
- Relative addressing
  - レジスタは“r0”のようにも書けるし, “r[0]”とも書いてよい
  - 後者の場合には、”r[数字|式]”と書ける
    - 例: mov r[1], r[2+10]
- b#とa#とaL:???

レジスタ(2)

- 特殊レジスタ
  - vWinCoord : Window Coordinate Rgeister
    - Index pairがはいっていると思えばよい
    - “vWinCord.x”がx座標
    - “vWinCord.y”がy座標
  - v#はimport register(入力レジスタ)
  - o#はexport register(出力レジスタ)
  - g[address]はglobalレジスタ
    - グローバル変数にアクセスするときに利用

プログラム例 (1)

```
1:  il_ps_2_0
2:  dcl_input_interp(linear) v0.xy
3:  dcl_output_generic o0
4:  dcl_cb cb0[1]
5:  dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
6:  sample_resource(0)_sampler(0) r0, v0.xyxx
7:  mul o0, r0, cb0[0]
8:  ret_dyn
9:  end
```

- 1行目
  - ILプログラムのバージョン表す文字列。決まり文句
- 2-5行目は変数の宣言文
  - 2行目: index pairを"v0.xy"として宣言
  - 3行目: 出力メモリを"o0"として宣言
  - 4行目: 定数メモリを1変数(4要素)宣言
  - 5行目: 入力メモリ(resource)の宣言。idは0番
    - メモリは2次元の配列

プログラム例 (2)

```
1:  il_ps_2_0
2:  dcl_input_interp(linear) v0.xy
3:  dcl_output_generic o0
4:  dcl_cb cb0[1]
5:  dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
6:  sample_resource(0)_sampler(0) r0, v0.xyxx
7:  mul o0, r0, cb0[0]
8:  ret_dyn
9:  end
```

- 6行目: resource 0からデータを読み込み
  - “v0.xy”のindexにあるメモリをr0レジスタに読み込む
  - resource 0は5行目で定義されている
- 7行目: 演算処理
  - r0とcb0[0](定数メモリ)を乗算してo0に保存
  - “o0”は3行目で出力メモリとして宣言されているので、結果は所定のindexの位置に保存される
- 8行目: kernelプログラムの終了
- 9行目: kernelプログラム記述を終える決まり文句

プログラム例 (3)

- このkernelは、2次元メモリに定数メモリの値を乗算して、結果を出力用の2次元メモリに保存する。
- 入力メモリ部分のプログラム抜粋

```
// 256x256の2次元メモリを確保する
calResAllocLocal2D(&inputRes, device, 256, 256, CAL_FORMAT_FLOAT_1, 0);
//初期値をセット
calCtxGetMem(&inputMem, ctx, inputRes);
calResMap((CALvoid**)&fdata, &pitch, inputRes, 0);
for (int i = 0; i < 256; ++i) {
    float* tmp = &fdata[i * pitch];
    for (int j = 0; j < 256; ++j) {
        tmp[j] = (float)(i * pitch + j);
    }
}
calResUnmap(inputRes);
// IL記述の変数をinputResと束縛。"i0"がIL記述でのinput resouce 0に対応する
calModuleGetName(&inName, ctx, module, "i0");
calCtxSetMem(ctx, inName, inputMem);
```

プログラム例 (4)

- 定数メモリ部分のプログラム抜粋

```
// 定数メモリを確保する
calResAllocRemote1D(&constRes, &device, 1, 1, CAL_FORMAT_FLOAT_4, 0);
//値をセット
calCtxGetMem(&constMem, ctx, constRes);
calResMap((CALvoid**)&constPtr, &constPitch, constRes, 0);
constPtr[0] = 0.5f,      constPtr[1] = 0.0f;
constPtr[2] = 0.0f;      constPtr[3] = 0.0f;
calResUnmap(constRes);
// IL記述の変数をconstResと束縛。IL記述の“cb0”を指定
calModuleGetName(&constName, ctx, module, "cb0");
```

- ホストのプログラムとIL記述との対応
  - input : “resource_id(#)”が”i#”
    - CAL_FORMATにより要素数が決まる
  - outpput : “output_generic o#”が”o#”
  - const : ILで宣言したそのまま(ただし配列)

様々な命令：変数宣言(1)

- dcl_cb cb#[n]
  - 定数メモリを宣言する。cb0-cb14の15個
  - 一つの要素は4-vector
  - 合計で4096個まで利用可能
- dcl_input vreg.xy
  - 入力レジスタを宣言
  - vレジスタでなくてはならない
  - vreg.xとvreg.yにindex pairが保持される

様々な命令：変数宣言(2)

- dcl_output_usage(X) oreg
  - 出力レジスタを宣言
  - 通常は"_usage(generic)"
- dcl_literal reg, <x>, <y>, <z>, <w>
  - 即値を宣言
  - <x>, <y>, <z>, <w>にhex値または浮動小数点値を書くことで、"reg.x"等でアクセスできる

様々な命令:変数宣言(3)

- dcl_resouce_id(#)_type(X)_fmtx(fmt)…
  - 入力メモリを宣言
  - “id(#)”にてresourceの番号を指定
    - メインプログラムの”calModuleGetName()”での指定は”i#”になる
  - “_fmtx(fmt)”にて変数形式を指定
    - 同様にy,z,wの形式も指定する
  - “sample_resource(#)_sampler(#) reg, src”で、宣言したメモリの内容をregに読み込み。srcレジスタのxyが読み込むメモリのindexになる
    - mem[3][4]を読み込む場合には、src.x = 3.0, src.y = 4.0とすればよい

様々な命令:算術命令(1)

- 四則演算と数学関数が用意されている
  - 四則演算は3 opperands形式
    - mul src, dst0, dst1 → src = dst0*dst1
    - 通常は、xyzwのcomponentごとのベクタ演算
  - ILは型のないアセンブラ言語であるため、命令により変数の型が規定される
    - 単精度浮動小数点型命令
      - mul src, dst0, dst1
    - 整数型命令
      - imul src, dst0, dst1
    - 倍精度浮動小数点型命令
      - dmul src, dst0, dst1
      - 倍精度の場合にはsrc, dstともxy componentのみ

様々な命令：算術命令(2)

- maskの利用
  - mul r0.x, r1.x, r2.x
    - x componentのみのスカラ計算
  - add r0.xyz, r1.xyz, r2.xyz
    - xyz componentのみのベクタ計算
- 特殊な命令の例
  - “rsq dst, src0”は$x^{-0.5}$逆数を計算するが、componentごとではなく、src0.wの逆数を計算して、dst.xyzwに結果を代入する
  - “rsq_vec dst, src0”命令でcomponentごとの計算になる

様々な命令:変換命令

- ftoi dst, src0 (単精度から整数)
  - “ftou”は単精度から符号なし整数
- itof dst, src0 (整数から単精度)
  - “utof”は符号なし整数から単精度
- d2f dst, src0 (倍精度から単精度)
- f2d dst, src0 (単精度から倍精度)

様々な命令：比較命令

- eq, ge, lt, ne (浮動小数点比較)
  - 例: eq dst, src0, src1
    - componentごとに比較の結果が成立したら、dstの対応する場所にTRUEが代入される

Table 6-7. Float Comparison Instructions

Function	Opcode	Text Instruction Syntax	Description
==	IL_OP_EQ	`eq dst, src0, src1`	TRUE if *src0* is equal to *src1*.
>=	IL_OP_GE	`ge dst, src0, src1`	TRUE if *src0* is greater-than or equal to *src1*.
<	IL_OP_LT	`lt dst, src0, src1`	TRUE if *src0* is less-than *src1*.
!=	IL_OP_NE	`ne dst, src0, src1`	TRUE if *src0* is not equal to *src1*.

- ieq, ige, ilt, ine (符号あり整数比較)
- uge, ult (符号なし整数比較)

様々な命令：制御構造

- ifc-else-endif
- whileloop-endloop
- switch-case-end
- continueとbreak
- if文の例
  - ifc_relop(eq) src0, src1
    - src0.xとsrc1.xが等しいならば次の文を実行
    - そうでない場合には、else節またはendif以降を実行
  - ifc_logicalz src0.x
    - src0.xが整数値0ならば次の文を実行

プログラム:N体の重力計算(1)

- 既存のアルゴリズムやアプリケーションをCALで実装するには
  - 前提として、並列計算可能な問題でなくては、利用する意味がない
  - GPUのアーキテクチャにあわせて、アルゴリズムを変更する必要あり
  - GPUのメモリに合わせた、効率のよいデータ構造を考える必要あり
  - 以下、例題として、重力多体問題の計算を実装した場合を説明

プログラム:N体の重力計算(2)

- 計算すべきモノ: 重力加速度とポテンシャル

$$\vec{a} = \sum \frac{m_j(\vec{r_i} - \vec{r_j})}{(|\vec{r_i} - \vec{r_j}|^2 + \varepsilon^2)^{3/2}}$$

$$p = \sum \frac{m_j}{(|\vec{r_i} - \vec{r_j}|^2 + \varepsilon^2)^{1/2}}$$

- 入力: 位置ベクトルと質量
  - ベクトル 3成分+スカラー 1
- 出力: 加速度ベクトルとポテンシャル
  - ベクトル 3成分+スカラー 1

プログラム:N体の重力計算(3)

- データ構造 (array of structure)
  - 入力: (x,y,z,w)の4成分を1粒子にわりあて
    - 単精度浮動小数点変数を利用
    - ホスト上 x[N], y[N], z[N], m[N]
    - GPUメモリ上

  - 出力: (x,y,z,w)の4成分を1粒子にわりあて
    - ホスト上 ax[N], ay[N], az[N], p[N]
    - GPUメモリ上 (入力と同様)

プログラム:N体の重力計算(4)

- ループ計算の詳細

1. r_i (x,y,z,m)を読み込む(1回のみ)
2. r_j を読み込む
3. r_iとr_jの間の相互作用を計算する

$$\vec{a} = \sum \frac{m_j(\vec{r_i} - \vec{r_j})}{(|\vec{r_i} - \vec{r_j}|^2 + \varepsilon^2)^{3/2}}$$

4. 結果を積算する

以上の2から4の繰り返し

プログラム:N体の重力計算(5)

- ループプによる計算部分

```
whileloop
  ige r88.x___, r100.x, r77.x
  break_logicalnz r88.x

  sample_resource(0)_sampler(0) r0, r2
  sub r5.xyz, r0.xyz, r4.xyz
  dp4 r6, r5, r5
  rsq r7, r6
  mul r8, r7, r7.xyz1
  mul r8, r8, r7.xyz1
  mul r9, r8, r5.xyz1
  mad r3, r9, r0.w, r3

  add r2.x___, r2.x, l1.x
  iadd r100.x___, r100.x, l0.z
  umod r101.x, r100.x, r77.y
  if_logicalz r101.x
    add r2.0y, r2.0y, l1.x
  endif
endloop
```

loop終了判定

相互作用計算

index更新

プログラム:N体の重力計算(6)

- 変数定義
  - r77.x: loop回数
  - r100.x: loop counter
  - r2.xy: rjを読み込むポインタ
  - r4: r_i
  - r0: r_j
  - r3: 結果(加速度とポテンシャル)積算変数
- 前処理
  - r5.wにεをいれておく
  - r2.xy(rjポインタ)を(0.0, 0.0)にリセット

プログラム:N体の重力計算(7)

- 1次元ループの例

```
whileloop
  ige r88.x___, r100.x, r77.x
  break_logicalnz r88.x

  ... loop body ....

  iadd r100.x___, r100.x, l0.z
endloop
```

- r100.xをloop couterとして
- loopごとにインクリメントする
- loopの先頭でloop回数と比較
  - loop回数を超えていたらwhileloopをbreakする
  - ILのwhilelopは明示的にbreakしない限り無限ループ

プログラム:N体の重力計算(8)

- 相互作用計算の注意点
  - dp4による $|\vec{r_i} - \vec{r_j}|^2 + \varepsilon^2$ の計算
  - rsqによる $x^{-0.5}$ の計算
  - r2.xyによるデータ読み込み
    - GPUメモリを2次元メモリとして定義しているので、r2.xyを正しくupdateする必要がある
    - x,yのupdate pattern (domain = {10,10})

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9

粒子メモリの構造

プログラム:N体の重力計算(9)

IL プログラム全体

```
il_ps_2_0
dcl_input_interp(linear) v0.xy
dcl_output_generic o0
dcl_cb cb0[1]
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
dcl_literal l0, 1.0, 0x0, 0x1, 1.0

mov r100.0, r100
mov r2.00, r2
mov r3.0000, r3
ftoi r77.xy, cb0[0].zw
mov r5.___w, cb0[0].y

sample_resource(0)_sampler(0) r4, v0.xy
whileloop
  ige r88.x___, r100.x, r77.x
  break_logicalnz r88.x

  sample_resource(0)_sampler(0) r0, r2
  sub r5.xyz, r0.xyz, r4.xyz
  dp4 r6, r5, r5
  rsq r7, r6
  mul r8, r7, r7.xyz1
  mul r8, r8, r7.xyz1
  mul r9, r8, r5.xyz1
  mad r3, r9, r0.w, r3

  add r2.x___, r2.x, l0.x
  iadd r100.x___, r100.x, l0.z
  umod r101.x, r100.x, r77.y
  if_logicalz r101.x
    add r2.0y, r2.0y, l0.x
  endif
endloop
mov o0, r3
ret_dyn
end
```

プログラム:N体の重力計算(10)

- ホストプログラムの流れ
  1. 初期化
     - データ読み込み
     - GPUメモリの設定
  2. 座標データの送信
  3. GPU kernelの実行
  4. kernel終了待ち
  5. 結果を回収
  6. 積分して新しい座標を得る
- 以上の2から6を繰り返し実行

プログラム:N体の重力計算(11)

- ホストプログラム抜粋

```
float* fdata = NULL;
CALuint pitch = 0;
CALdomain domain = {0, 0, nx, ny};
CALevent e = 0;

// データの送信: 粒子データ(X[N][4]という2次元配列)からindataにコピー
calResMap((CALvoid**)&fdata, &pitch, indata, 0);
int k = 0;
for (int j = 0; j < ny; ++j) {
  float *tmp = &fdata[j*pitch*4];
  for (int i = 0; i < nx; ++i) {
    float *cur = &tmp[i*4];
    memcpy(cur, X[k++], 4*sizeof(float));
  }
}
calResUnmap(indata);

// kernel の実行と終了待ち
calCtxRunProgram(&e, cc, ff, &domain);
while (calCtxIsEventDone(cc, e) == CAL_RESULT_PENDING);

// 結果を回収: 加速度とポテンシャルをA[N][4]という2次元配列にコピー
calResMap((CALvoid**)&fdata, &pitch, outdata, 0);
memcpy(A, fdata, nbody*4*sizeof(float));
calResUnmap(outdata);
```

プログラム:N体の重力計算(12)

- 性能の測定
  - ボードのピーク性能: ~ 1 TFLOPS
    - 160 (VLIW units) x 5 (FP units) x 2 (FPMAD) x 650 (MHz) ~ 1040 GFLOPS
    - 積和演算(FPMAD)
      - $A = \alpha B + C$ (2浮動小数点演算)
  - 1 interaction 21演算でcountした実測値
    - ~ 100 GFLOPS
    - 全命令で積和演算の活用は不可能
    - 4要素の演算になっていない部分がある

プログラム:N体の重力計算(13)

- 高速化の余地
  - 今の実装は、array of structureを利用
    - 個々のkernelが1粒子を担当
    - 1粒子読み込み、1粒子の計算
      - one iterationで1相互作用を計算
      - 計算数/メモリの比率が小さい
    - 4要素のベクトルSIMD命令が効率的でない
  - Structure of Arrayにすると
    - 個々のkernelが4粒子を担当
    - 4粒子読み込みして、4 x 4 = 16相互作用
      - 計算数/メモリの比率が大幅増加
    - ベクトルSIMD命令が少し効率的になる