DS-CUDA ソフトウェアパッケージ ユーザガイド

for DS-CUDA version 2.5.0

最終更新：2015 年 8 月 5 日

川井 敦
E-mail: kawai@kfcr.jp

目 次

1　本文書の概要

この文書では DS-CUDA ソフトウェアパッケージの使用方法を説明します。DS-CUDA は PC の I/O スロットに接続された NVIDIA 社製 GPU カード (CUDA デバイス) を、ネットワーク接続された他の PC からシームレスに使用するためのミドルウェアです。本バージョンは CUDA バージョン 7.0 で動作確認ずみですが、CUDA バージョン 7.0 の提供するすべての機能をサポートしているわけではありません (節 5.1 参照)。

以降、第 2 章では DS-CUDA の基本構成と動作概要を説明します。第 3 章では DS-CUDA ソフトウェアパッケージ (以降「本パッケージ」と呼びます) のインストール方法を説明します。第 4 章ではアプリケーションプログラムのコンパイル方法、実行方法について説明します。第 5 章では DS-CUDA の実装や内部動作について触れます。

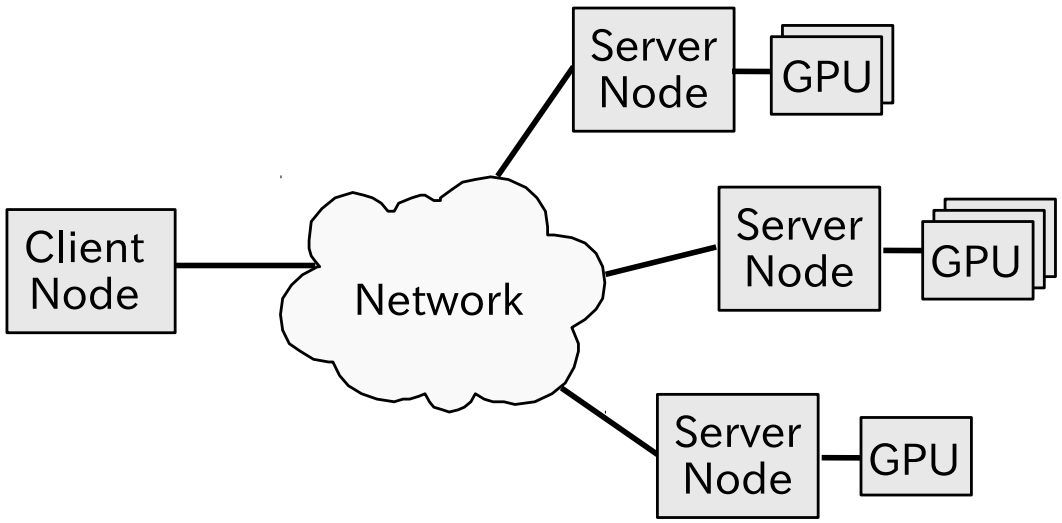
なお以降では、本パッケージのルートディレクトリ (/パッケージを展開したディレクトリ/dscudapkg バージョン番号/) を $dscudapkg と表記します。

2　DS-CUDA 概観

本節では DS-CUDA の基本構成と機能を概観します。

2.1　機能概要

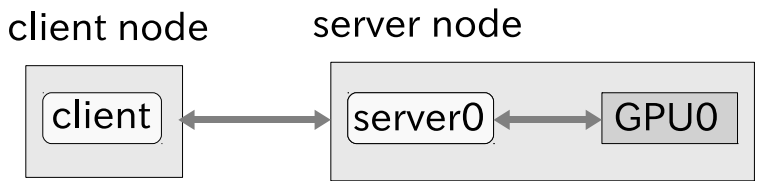
DS-CUDA を用いると、アプリケーションプログラムはネットワーク上に分散配置された GPU を透過的に扱えます。例えば下図の Client Node 上のアプリケーションは、Client Node 自身にローカルにインストールされた GPU へアクセスする場合と同じプログラミングインタフェースを用いて Server Node に接続された GPU へアクセスできます。



図：GPU クラスタの例。

2.2 システムの構成

もっとも単純な例として、PC 2 台と GPU 1 台からなるシステムを考えます。



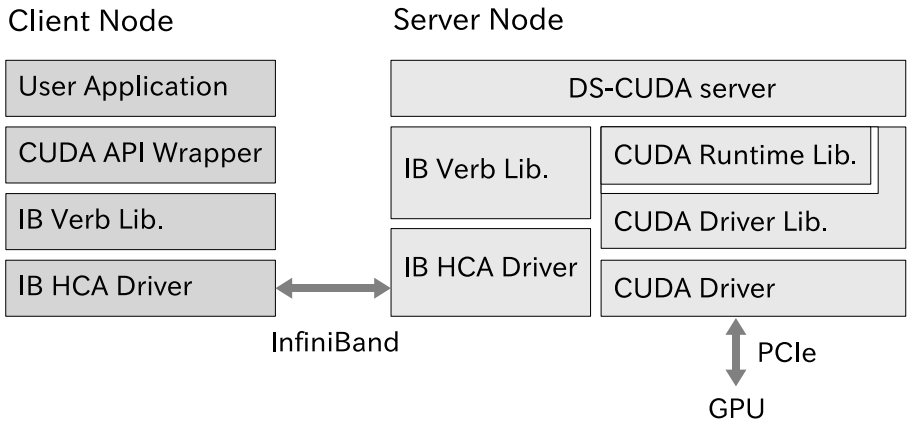
図：最小構成の DS-CUDA システム。

このシステムは互いにネットワーク接続された 2 台の PC と、その一方にインストールされた GPU から構成されます。GPU は NVIDIA 社の CUDA に対応した製品 (CUDA デバイス) であることが必須です。ネットワーク接続には原則として InfiniBand の使用を想定します。ただし TCP/IP による通信が可能なネットワークであれば、InfiniBand 以外のものも使用できます (例: GigabitEthernet)。簡便のため、GPU を持つ側の PC を server node と呼び、もう一方の PC を client node と呼ぶことにします。

DS-CUDA はユーザに対し、クライアント・サーバ型の実行環境を提供します。serve node 上ではサーバプログラムを常時稼働させておき (図中の server0)、client node 上のアプリケーションプログラム (図中の client) をサーバプログラムに対するクライアントとして実行します。サーバプログラムはアプリケーションプログラムの要求に従って GPU を制御します。

2.3 ソフトウェア階層

DS-CUDA のソフトウェアは下図に示すように階層化されています。



図：ソフトウェア階層

client node 上のユーザアプリケーションは CUDA API を用いて GPU (CUDA デバイス) にアクセスします。内部的にはこれは server node 上のサーバプログラムへのアクセスに置き換えられますが、ユーザアプリケーションはそのことを意識する必要はありません。クライアント・サーバ間の通信プロトコルには InfiniBand Verb もしくは TCP/IP を使用します (図のソフトウェア階層は InfiniBand Verb を使用した場合のもの)。

2.4 クライアントプログラムの生成

CUDA C/C++ で記述したユーザアプリケーションのソースコードを、本パッケージの提供する DS-CUDA プリプロセッサ `dscudacpp` を用いてコンパイルすることにより、client node で動作するプログラムを生成します。

2.5 冗長デバイス

DS-CUDA は冗長デバイスを扱えます。冗長デバイスとは複数の GPU を用いて構成された単一の仮想デバイスです。この仮想デバイス上で計算を実行すると、仮想デバイスを構成する複数の GPU 上で同一の計算が行われ、両者の結果が異なっていた場合には、その旨がユーザアプリケーションに通知されます。

アプリケーションプログラムが一定の制約 (節 4.4 参照) を満たすように記述されている場合には、誤りを検出した計算を自動的に再実行させることも可能です。

3 インストール

3.1 準備

本パッケージは以下のソフトウェアに依存しています。インストール作業の前に、これらの動作環境を整えて下さい。

- CUDA 開発ツール (CUDA 7.0 で動作確認済)
  `http://www.nvidia.com/`
- C++ コンパイラ (g++ version 4.4.6 で動作確認済)
  `http://gcc.gnu.org/`
- Ruby (version 1.8.7 で動作確認済)
  `http://www.ruby-lang.org/`
- OFED (version 3.5 で動作確認済)
  `https://www.openfabrics.org/resources/\`
  `ofed-for-linux-ofed-for-windows/linux-sources.html`

注意 :

- コンパイル対象とするアプリケーションプログラムが CUDA カーネルを C++ テンプレートとして実装している場合には、C++ コンパイラには g++ version 4.0.0 以上を使用してください。それ以前のバージョンや、Intel C++ コンパイラ等では動作しません。これは C++ テンプレートからシンボル名を生成する際の name mangling 規則がコンパイラごとに異なっており、DS-CUDA では現在のところ g++ version 4 系の name mangling 規則のみをサポートしているためです。
- ruby は `/usr/bin/` にインストールして下さい。他のディレクトリにインストールされている場合には `/usr/bin/` へシンボリックリンクを張って下さい。
- `/etc/security/limits.conf` に以下の 2 行を追記して下さい。

  ```
  * hard memlock unlimited
  * soft memlock unlimited
  ```

  この設定は root 権限を持たない一般ユーザが InfiniBand Verb 経由の通信を行うために必要です。

3.2 パッケージの展開

ソフトウェアパッケージ `dscudapkg`n`.tar.gz` を展開してください (n はバージョン番号)。パッケージには以下のファイルが含まれています:

doc/	本文書、その他のドキュメント。
scripts/	パッケージ管理ユーティリティ。
bin/	
dscudacpp	.cu ファイルから DS-CUDA クライアントを生成するプリプロセッサ。
pretty2mangled	CUDA カーネルの name mangling されたシンボル名を取得するスクリプト。`libdscudasvr.a` が使用します。
dscudad	DS-CUDA デーモン。
include/	ヘッダファイル (DS-CUDA クライアント・サーバ共用)。
lib/	
libdscuda_ibv.a	DS-CUDA クライアントライブラリ (InfiniBand Verb インタフェース)。
libdscuda_tcp.a	DS-CUDA クライアントライブラリ (TCP/IP インタフェース)。
libdscudasvr.a	DS-CUDA サーバライブラリ。
libdscudart.so	CUDA ランタイムライブラリのダミー。
src/	DS-CUDA ライブラリ群のソースコード。
sample/	アプリケーションプログラムの例。

3.3 環境変数の設定

以下の環境変数を設定してください。

client node, server node 共通	
CUDAPATH :	CUDA Toolkit のインストールされているパス。 デフォルト値は `/usr/local/cuda`
CUDASDKPATH :	CUDA SDK のインストールされているパス。 デフォルト値は `/usr/local/cuda/samlles`
DSCUDA_PATH :	DS-CUDA ソフトウェアパッケージのインストールされて いるパス。設定必須。デフォルト値はありません。
DSCUDA_WARNLEVEL :	DS-CUDA サーバおよびクライアント実行時のメッセージ 出力レベル。整数値を指定します。値が大きいほど詳細なメッ セージが出力されます。デフォルト値は 2、最小値は 0 です。
DSCUDA_SVRPATH :	DS-CUDA サーバプログラムの実行ファイルがおかれている client node 上のパス、およびクライアントプログラム起動時に client node からコピーされる server node 上のパス。
DSCUDA_USEGD2 :	値 1 を設定すると GPU Direct ver2 を使用します (節 4.7 参照)。
DSCUDA_USEGD3 :	値 1 を設定すると GPU Direct ver3 を使用します (節 4.7 参照)。
server node のみ	
DSCUDA_REMOTECALL	通信プロトコルを選択します。指定できる値は `ibv`, `tcp` のいずれかです。それぞれ InfiniBand Verb, TCP を意味します。 DS-CUDA サーバの起動が DS-CUDA デーモンを介して行われる 場合には、通信プロトコルは自動的に選択され、この変数の値は 無視されます。
client node のみ	
LD_LIBRARY_PATH :	共有ライブラリパスに `$DSCUDA_PATH/lib` を追加してください。 設定必須。
DSCUDA_SERVER :	DS-CUDA サーバが動作している PC の IP アドレス、 あるいはホスト名。デフォルト値は `localhost` 複数のサーバを使用する場合の記法については 節 4.3.2 を参照して下さい。

(次ページへ続く)

(前ページより続く)

DSCUDA_SERVER_CONF :	DS-CUDA サーバが動作している PC の IP アドレス、あるいはホスト名を記述したファイルのファイル名。環境変数 DSCUDA_SERVER への設定値が長く煩雑になってしまう場合 (多数のサーバを使用する場合など) 、設定値をファイルに記述し、そのファイル名をこの環境変数で指定できます。
DSCUDA_AUTOVERB :	冗長デバイス上で自動再計算機能を使用する場合にこの変数を定義します。変数にはどのような値を設定しても構いません。
DSCUDA_CP_PERIOD :	自動再計算のためのチェックポイントを設定する時間間隔を指定します。単位は秒、デフォルト値は 60 です。
DSCUDA_SERVER_SPARE:	ライブマイグレーション用の予備サーバの IP アドレス、あるいはホスト名 (節 4.5 参照)。
DSCUDA_USEDAEMON :	DS-CUDA サーバの起動を DS-CUDA デーモンを介して行う場合にこの変数に値 1 を定義します。

例:

```
kawai@client>export DSCUDA_PATH="/home/kawai/src/dscudapkg2.0.0"
kawai@client>export DSCUDA_SERVER="192.168.10.101"
kawai@client>export LD_LIBRARY_PATH=/home/kawai/src/dscudapkg2.0.0/lib:\
$LD_LIBRARY_PATH
```

CUDA や C コンパイラが参照する環境変数がある場合には、必要に応じてそれらも設定して下さい。

3.4 ライブラリ・実行ファイルの生成

ディレクトリ $dscudapkg/src へ移動し、make を実行してください。以下のファイルが生成されます。

- DS-CUDA クライアントライブラリ (IBV インタフェース): $dscudapkg/lib/libdscuda_ibv.a
- DS-CUDA クライアントライブラリ (TCP インタフェース): $dscudapkg/lib/libdscuda_tcp.a
- CUDA ランタイムライブラリのダミー: $dscudapkg/lib/libcudart.so
- DS-CUDA サーバライブラリ: $dscudapkg/lib/libdscudasvr.a
- DS-CUDA デーモン: $dscudapkg/bin/dscudad

3.5 動作チェック

$dscudapkg/sample/ 内のサンプルプログラムを使用して、本パッケージの動作を確認します。

3.5.1 サンプルプログラム

$dscudapkg/sample/ 内に各種のサンプルプログラムが格納されています。

- vecadd: ベクトルの加算を行います。
- vecadd_cmem: ベクトルの加算を行います。コンスタントメモリを使用します。
- direct: 重力多体シミュレーションを行います (make run で初期条件を生成し、シミュレーション実行します)。
- claret: 溶融塩のシミュレーションを行います。
- bandwidth: ホストとの通信速度を測定します。
- p2p: デバイス間通信を行います。
- reduction: 複数デバイスにまたがる reduction を行います。
- cdpSimpleQuicksort: NVIDIA 社の提供するクイックソートのサンプルコードです。DS-CUDA 向けに Makefile を変更してあります。
- cdpAdvancedQuicksort: NVIDIA 社の提供するクイックソートのサンプルコードです。DS-CUDA 向けに Makefile を変更してあります。
- exafmm: ExaFMM (横田理央氏開発の FMM コード) です。
  https://bitbucket.org/rioyokota/exafmm-dev の 2014/07/01 版をベースに、DS-CUDA 上で動作するよう Makefile およびソースコードを変更してあります。

各ディレクトリ内で make を実行すると、それぞれの DS-CUDA クライアントとサーバが生成されます (サーバのファイル名は、クライアントのファイル名の末尾に .svr を付与したものとなります)。

4 使用方法

4.1 基本的な手順

従来の DS-CUDA (バージョン 2.0.0 よりも前) では、以下の手順でアプリケーションプログラムを生成、実行していました。

1) アプリケーションプログラムのソースコードを dscudacpp を用いてコンパイルし、クライアント (実行ファイル) とデバイスコード (PTX データ) を生成する。

2) server node 上で DS-CUDA デーモンを起動する。

3) client node 上で クライアントを起動する。

4) クライアントからの要求に応じて DS-CUDA デーモンが DS-CUDA サーバを起動する。

5) クライアントと DS-CUDA サーバが通信を確立し、処理をすすめる。

CUDA カーネル (デバイスコード) は、クライアントの実行時に client node から server node へ転送されていました。より具体的には、client node 上の PTX データ) がクライアントからサーバへ転送され、サーバはこのデータを動的にロード、実行していました。

この方法は簡便で洗練されていますが、Dynamic Parallelism を使用するデバイスコードを扱えません。デバイスコードの動的ロードには CUDA ドライバ API cuModuleLoadData() を使用しますが、この API は Dynamic Parallelism をサポートしていないためです。

DS-CUDA バージョン 2.0.0 では、簡便さを犠牲にして Dynamic Parallelism をサポートしました。バージョン 2.0.0 以降は以下の手順でアプリケーションプログラムが実行されます。

1) アプリケーションプログラム のソースコードを dscudacpp を用いてコンパイルし、クライアント、サーバそれぞれの実行ファイルを生成する。

2) server node 上で DS-CUDA デーモンを起動する。

3) client node 上でクライアントを起動する。

4) クライアントは自身に対応するサーバのイメージを DS-CUDA デーモンへ送信する。

5) DS-CUDA デーモンはクライアントからサーバのイメージを受信し、それをファイルとして保存し、起動する。

6) クライアントとサーバが通信を確立し、処理をすすめる。

この方法ではアプリケーションごとに専用のサーバが生成されます。生成されたサーバのイメージはクライアント起動時に server node へ送信されます。

従来にくらべて余分な準備手続きを必要としますが、Dynamic Parallelism を使用するコードも扱えるという利点があります。CUDA カーネル (デバイスコード) はサーバイメージに含まれており、サーバの実行中に動的にロードする必要がありません。そのため CUDA ドライバ API が Dynamic Parallelism をサポートしていないことは問題になりません。

4.2 アプリケーションプログラム の生成

4.2.1 一般的な生成方法

CUDA C/C++ で記述されたユーザアプリケーションのソースコード (以下.cu ファイルと表記します) から DS-CUDA クライアントを生成するには、DS-CUDA プリプロセッサ $dscudapkg/bin/dscudacpp を使用します。

dscudacpp を引数を与えずに実行すると、使用方法が表示されます。

```
kawai@client>pwd
/home/kawai/src/dscuda2.0.0/sample/direct
kawai@client>../../bin/dscudacpp
No input file given.
  usage: ../../bin/dscudacpp [options] inputfile(s)...
  options:
      --infile <file>   : a .cu input file.
      -i <file>

      -o <file>         : an output file.

      --verbose[=level] : be verbose. the level can optionally be given. [2]
      -v[level]           the higher level gives the more verbose messages. \
  level 0 for silence.

      --help            : print this help.
      -h
  Note that all options not listed above are implicitly passed on to nvcc.
```

dscudacpp へ入力ファイル (.cu .c .o など) を与えるには、オプションスイッチ -i を使用します。また、生成される DS-CUDA クライアントは、オプションスイッチ -o で指定します。これら以外のすべての引数やオプションスイッチは、dscudacpp では解釈されずに、dscudacpp が内部的に起動する nvcc へと渡されます。nvcc が必要とする引数は、すべて dscudacpp への引数として渡さねばなりません。

dscudacpp に -c オプションを与えてオブジェクトファイルを生成しようとした場合には、クライアント向けの通常のオブジェクトファイル (拡張子 .o) に加えて、サーバ向けのオブジェクトファイル (拡張子 .svr.o) も同時に生成されます。

```
kawai@client>dscudacpp -c -o foo.o -i foo.cu
kawai@client>ls
dscudatmp/  foo.cu  foo.o  foo.svr.cu  foo.svr.o
```

dscudacpp に -link オプションを与えて (あるいはフェーズを指定するオプションを何も与えずに) 実行ファイルを生成しようとした場合には、client node 上で動作するクライアント (TCP インタフェース用と IBV インタフェース用の 2 種類の実行ファイル) と、

server node 上で動作するサーバが同時に生成されます。サーバのファイル名はクライアントのファイル名の末尾に.svr を付与したものとなります。

```
kawai@client>dscudacpp -o foo -i foo.cu bar.cu
kawai@client>ls
dscudatmp/  bar.svr.cu  bar.o        foo_ibv      foo_tcp  foo.cu     foo.svr.cu
bar.cu      bar.svr.o   foo_ibv.svr  foo_tcp.svr  foo.o    foo.svr.o
```

4.2.2 やや特殊な状況の扱いなど

main() 関数の扱い

dscudacpp がサーバを生成する際に使用する main() 関数は libdscudasvr.a 内に含まれています。いっぽう、アプリケーションの実行ファイルを生成する際に使用する main() 関数は、アプリケーションのソースコード内で定義されています。両者の main() 関数の衝突を避けるために、dscudacpp はアプリケーションのソースコード内の ‘‘main(’’ という文字列を別の文字列に置換してからコンパイルを行います。

この文字列置換処理がプログラムに副作用をもたらす場合 (例えばソースコード内に ‘‘main(’’ という文字列リテラルを含む場合など) には、dscudacpp に --prm オプションを与えて置換機能を無効化してください。このオプションを使用する場合には、main() 関数の衝突を避けるために、ソースコード中の main() 関数の定義部を手動で下記のように変更してください。

例:

```
#ifdef __DSCUDA_SERVER__
int main(int argc, char **argv)
#else
int unused_main(int argc, char **argv)
#endif
{
  ....
}
```

ここで__DSCUDA_SERVER__ は dscudacpp がサーバをコンパイルする際に自動的に定義する定数マクロです。

マクロ定数

dscudacpp は以下の C プリプロセッサマクロ定数を定義します。定数はソースコード中で参照できます。

定数名	値
__DSCUDA__	1
__DSCUDACPP_VERSION__	バージョン番号 (例：バージョン 1.2.3 の値は 0x010203)
__DSCUDA_SERVER__	1 サーバプログラムコンパイル時にのみ定義されます。
__DSCUDA_CLIENT__	1 クライアントプログラム コンパイル時にのみ定義されます。

通信ライブラリのリンクオプション

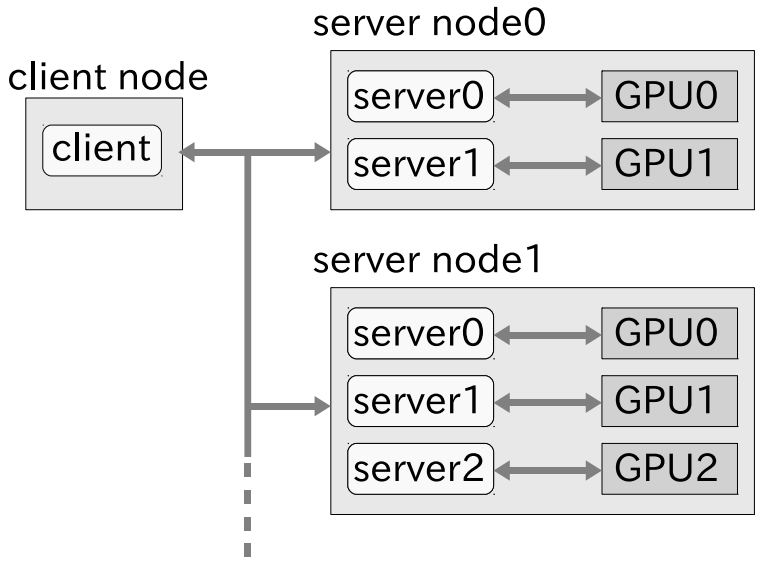
クライアント/サーバ間の通信に必要なライブラリは暗黙にリンクされますので、従来のバージョンで必要とされていた下記のオプションは不要になりました。

- 通信プロトコルとして InfiniBand Verb を用いる場合:
  `--cudart=shared -ldscuda_ibv -libverbs -lrdmacm -lpthread`
- 通信プロトコルとして TCP/IP を用いる場合:
  `--cudart=shared -ldscuda_tcp`

4.3 アプリケーションプログラムの実行

client node 上でアプリケーションプログラムを実行するには、server node 上での事前の設定が必要です。

またこれまでは簡単のために 1 台の GPU がインストールされた 1 台の server node のみを考えてきましたが、複数の server node を持つシステム (例えば下図) の場合には、client node のアプリケーションプログラムの使用する GPU を client node 側で指定する必要があります。



図：複数の GPU を持つシステムの例。

以下ではアプリケーションプログラムの実行に必要な、server node、client node 上の設定ついて説明します。

4.3.1 Sever node の設定

DS-CUDA デーモンの起動 ：client node が使用するすべての server node 上で、DS-CUDA デーモン `dscudad` を起動します。デーモンはクライアントからの要求に応じて動的にサーバを起動します。デーモンの実行ファイルは `$dscudapkg/bin/dscudad` です。これをコマンドラインから実行します。1 台の server node に複数の GPU がインストールされている場合でも、実行するデーモンは 1 node 当り 1 つだけです。

DS-CUDA サーバのパス指定 ：これから実行しようとするアプリケーションプログラムのサーバは、デーモンがクライアントの起動時に client node から受信し、server node 上の任意のディレクトリに配置します。配置先ディレクトリのフルパス名を、環境変数 `DSCUDA_SVRPATH` に設定します。

例:

```
kawai@server>export DSCUDA_SVRPATH="/home/kawai/var"
```

4.3.2 Client node の設定

client node 側は、クライアントがどの server node 上のどのサーバと通信するかを、環境変数 DSCUDA_SERVER によって指定します。DSCUDA_SERVER には以下の書式に従って記述した文字列を設定します。

$$server\_node{:}server\_id$$

ここで $server\_node$ は server node の IP アドレスあるいはドメイン名です。$server\_id$ はサーバプログラムの ID (節 4.3.1 参照) です。次の例のように複数のサーバプログラムを空白で区切って列挙すると、

```
192.168.0.100:0 192.168.0.100:1 192.168.0.105:0 192.168.0.107:1
```

クライアントからは複数の GPU があるように見えます。クライアントから見える仮想的な GPU (以降仮想デバイスと呼びます) のデバイス ID は、列挙した順に 0, 1, 2, ... が割り振られます。

冗長デバイス (節 2.5 参照) を構成するには、複数のサーバプログラムを空白ではなくカンマで区切って列挙します。例えば

```
192.168.0.100:0,192.168.0.100:1,192.168.0.105:0
```

は、server node 192.168.0.100 にインストールされた 2 枚の GPU と 192.168.0.105 にインストールされた 1 枚の GPU、合計 3 枚の GPU を用いて 1 台の冗長デバイスを構成します。

空白区切りとカンマ区切りを混在することも可能です。例えば

```
mysvr0:0 mysvr0:1,mysvr1:0 mysvr2:0
```

は、合計 4 台の GPU を用いて、通常の仮想デバイス 2 台と冗長デバイス 1 台、合計 3 台の仮想デバイスを構成します。

4.4 冗長デバイスによる誤り検出と自動再計算

冗長デバイス (節 2.5、節 4.3.2) を用いて計算を行うと、冗長デバイスを構成する複数の GPU 上で同一の計算が実行され、それらの結果が一致するかどうかが検証されます。一致しなかった場合には、いずれかの GPU で行われた計算の結果が誤っていたと見なされ、あらかじめ設定しておいたエラーハンドラが呼び出されます。

4.4.1 エラーハンドラの設定

エラーハンドラの設定には DS-CUDA の提供する API、dscudaSetErrorHandler() を用います。

書式:
void dscudaSetErrorHandler(void (*$handler$)(void *), void *$handler\_arg$)

引数 $handler$ に、エラーハンドラへのポインタを渡します。エラーハンドラは void * 型の引数をひとつ取れます。この引数を引数 $handler\_arg$ として与えます。引数が不要の場合には NULL を与えてください。

参考 : 同一のアプリケーションプログラムを、ソースコードを変更すること無く DS-CUDA プリプロセッサ dscudacpp と従来の CUDA コンパイラ nvcc の両方で処理できるようにするためには、以下に示すように C プリプロセッサディレクティブを用いて dscudaSetErrorHandler() の呼び出しを保護してください。

```
#ifdef __DSCUDA_CLIENT__
    dscudaSetErrorHandler(errhandler, (void *)&data);
#endif
```

ここで __DSCUDA_CLIENT__ は dscudacpp が自動的に定義する定数マクロです (節 4.2 参照)。

4.4.2 自動再計算

DS-CUDA には自動再計算機能が実装されています。つまり、計算結果に誤りが生じた場合にその計算を自動的に再実行させることが出来ます。GPU に恒久的な故障が生じた場合には再実行は無意味ですが、確率的にまれに生じる計算の誤りであれば、再実行によって訂正できる場合があります。

自動再計算機能を用いるには環境変数 DSCUDA_AUTOVERB を定義してください。変数は定義さえされていれば、値はどのような値を設定しても構いません。

ただし自動再計算は任意のアプリケーションプログラムで正しく機能するわけではありません。自動再計算機能は以下の手順で再計算を行います。

(1) クライアントの実行中は、CUDA API の呼び出しが行われるたびにその呼び出し履歴 (つまり API 名とすべての引数) を内部バッファに保存します (ただし現在のところ保存するのはホスト-GPU 間のデータ転送を行う CUDA API とカーネル実行のみ)。

(2) また cudaMalloc() によって確保したデバイスメモリのイメージを、定期的にメインメモリ上に保存します (以降ではこの機能をチェックポインティングと呼びます)。

チェックポインティング時にはこれまでの CUDA API の呼び出し履歴が不要になるため、履歴を消去します。

チェックポインティングの時間間隔は環境変数 DSCUDA_CP_PERIOD で指定する秒数で調整できます。すなわちデバイスからホスト方向への cudaMemcopy() によるデータ転送 (以降 D2H と表記します) を行う際に、前回の D2H からの経過時間が指定秒数を超えていた場合、チェックポインティングが発生します。

(3) GPU の計算結果を D2H によってホストへ回収する際に回収結果が誤っていれば、チェックポインティング時に保存したメモリイメージをデバイスメモリ上に復旧し、保存しておいた CUDA API の呼び出し履歴を順に再実行します。

自動再計算が正しく機能するためには、この手順で再計算を行った時に正しい結果を返すようにアプリケーションプログラムが記述されている必要があります。つまり GPU 上の計算結果が、直近のチェックポインティング時のメモリイメージと、その後の CUDA API 呼び出しだけに依存していることが必要です。

例えば GPU 上の特定の変数を毎ステップインクリメントさせるプログラムでは、自動再計算は正しく動作しません。再計算を行った時に、その変数が余分にインクリメントされてしまいます (チェックポインティング時に変数の値は保存されないため)。

4.5 サーバのライブマイグレーション

サーバとの通信が途絶えた場合、クライアントはそのサーバで行っていた計算を、予備サーバ上で続行しようとします。

予備サーバはあらかじめ、その IP アドレスもしくはホスト名を、環境変数 DSCUDA_SERVER_SPARE で指定しておきます。複数のサーバを指定するには空白で区切って列挙します。

サーバとの通信が途絶えると、クライアントは一台の予備サーバとの接続を試みます。接続に成功すると、直近のチェックポインティング時に保存しておいたメモリイメージをそのサーバのデバイスメモリ上に展開し、保存しておいた CUDA API の呼び出し履歴を順に実行します。これらが成功すると、その後はこのサーバを用いて計算を続行します。このようなサーバの引き継ぎをライブマイグレーションと呼ぶことにします。

4.6 独自の API

DS-CUDA には、オリジナルの CUDA API に加えていくつかの独自 API があります。

void dscudaMemcopies(void *****dbufs*, **void *****sbufs*, **int ****counts*, **int *ncopies*) : 複数 (ncopies 個) のデータ転送をまとめて実行します。i 個目のデータ転送の転送元アドレス、転送先アドレスは、それぞれ sbufs[i]、dbufs[i] で指定します。転送量は counts[i] にバイトサイズで指定します。転送元および転送先のデバイスは、アドレス

(UVA) から自動的に判定されます。同時実行可能なデータ転送は、複数のスレッド上で並列に実行されます。

void dscudaBroadcast(void **_dbufs_, void *_sbuf_, int _count_, int _ncopies_) : アドレス sbuf から ncopies 個のアドレスへの放送 (broadcast) を行います。i 個目の放送先のアドレスは dbufs[i] で指定します。転送量は count にバイトサイズで指定します。放送元および放送先のデバイスは、アドレス (UVA) から自動的に判定されます。放送はバイナリツリーネットワークによって実装されています。ネットワーク中の同時実行可能なデータ転送は、複数のスレッド上で並列に実行されます。

cudaError_t dscudaSortIntBy32BitKey(const int _size_, int *_key_, int *_value_) : 配列 value に格納された size 個の整数値を、対応する 32 ビットキー key の昇順にソートします。

cudaError_t dscudaSortIntBy64BitKey(const int _size_, uint64_t *_key_, int *_value_) : 配列 value に格納された size 個の整数値を、対応する 64 ビットキー key の昇順にソートします。

cudaError_t dscudaScanIntBy64BitKey(const int _size_, uint64_t *_key_, int *_value_) : 配列 value に格納された size 個の整数値を、対応する 64 ビットキー key で指定されるサブグループ単位でスキャンします。結果は value に返ります。cf. thrust::inclusive_scan_by_key()

例:

```
value[] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1,};
key[] = {0, 0, 0, 111, 111, 222, 333, 333, 333, 444};
```

に対して

```
dscudaScanIntBy64BitKey(10, key, valu);
```

を実行すると、value[] には

```
{1, 2, 3, 1, 2, 1, 1, 2, 3, 1}
```

が返ります。

4.7 GPU Direct への対応

4.7.1 GPU Direct とは

GPU Direct は GPU と他の PCI Express デバイスとのデータ転送の性能向上を目的として NVIDIA が提供している規格です。現時点では version 1, 2, 3 が規定されています。それぞれの機能は以下の通りです。

GPU Direct version 1: GPU とその他の PCI Express デバイス (ネットワークカード等) の間のデータ転送において、従来は両者がホスト計算機のメインメモリ上に別個のバッファを確保していました。そのためデータ転送時にはバッファ間のデータコピーが必要でした。GPU Direct verson 1 は両者が使用するバッファを共通化し、バッファ間のデータコピーを不要とします。この機能はメインメモリ上のバッファを (`malloc()` 等ではなく) `cudaMallocHost()` で pinned メモリとして確保した場合に使えます。CUDA 3.1 以降でサポートされています。

GPU Direct version 2: GPU 間の P2P データ転送は、従来はホスト計算機のメインメモリを介して行われていました。GPU Direct verson 2 は二つの GPU がメインメモリを介さず直接にデータ転送を行うことを可能とします。この機能は二つの GPU が共通の IO ハブに接続されている場合にのみ使えます。CUDA 4.0 以降でサポートされています。

GPU Direct version 3: GPU とその他の PCI Express デバイス (ネットワークカード等) の間のデータ転送は、従来はホスト計算機のメインメモリを介して行われていました。GPU Direct version 3 は二つのデバイスがメインメモリを介さず直接にデータ転送を行うことを可能とします。この機能を使うと、例えばあるホスト計算機に接続された GPU から別のホスト計算機に接続された GPU へ、それぞれのホスト計算機のネットワークカードがデータを直接転送できます。この機能は GPU と相手の PCI Express デバイスが共通の IO ハブに接続されている場合にのみ使えます。また GPU と相手の PCI Express デバイス双方のデバイスドライバに専用の機能拡張が必要です。CUDA 5.0 以降でサポートされています。

4.7.2 DS-CUDA の対応状況

DS-CUDA は GPU Direct に部分的に対応しています。対応状況は以下の通りです。なお通信プロトコルは InfiniBand Verb を使用することが前提です (TCP/IP は対応していません)。

GPU Direct version 1: 対応していません。

GPU Direct version 2: 対応しています。環境変数 `DSCUDA_USEGD2` の値を 1 に設定すると使用できます (デフォルトでは使用しません)。転送元と転送先の GPU が同一のサーバノードの共通の IO ハブに接続されている場合にのみ機能します。

GPU Direct version 3: GPU 間の P2P 転送についてのみ対応しています。ホスト計算機と GPU の間の転送については対応していません。環境変数 `DSCUDA_USEGD3` の値を 1 に設定すると使用できます (デフォルトでは使用しません)。動作させるためには InfiniBand HCA とそのデバイスドライバが GPU Direct version 3 に対応していることが必要です。また GPU のデバイスドライバに拡張モジュール (`nv_peer_mem`) が必要です。

以下の組み合わせで動作確認がとれています。

- GPU: NVIDIA Tesla K20c
- InfiniBand HCA: Mellanox MT27600 Connect-IB
- OS: CentOS 6.2 (`kernel2.6.32`)
- InfiniBand 環境: `MLNX_OFED_LINUX-2.4-1.0.0-rhel6.2-x86_64`
- GPU ドライバ拡張モジュール: `nv_peer_mem`

ただしデータ転送性能は GPU Direct を用いない場合よりも遅くなってしまいます。原因はいまのところ不明です。

5 DS-CUDA 実装の詳細

5.1 CUDA ランタイムライブラリのサポート範囲

DS-CUDA は CUDA ランタイムライブラリのすべての機能や API を仮想化するわけではありません。以下の条件に該当するものを含むいくつかの機能および API は、現在のところ仮想化されていません。

- グラフィクス制御を伴う API。
  例: `cudaGraphicsGLRegisterBuffer()`
- 非同期 API。
  例: `cudaMemcpyAsynch()`

5.2 CUDA C/C++ 文法のサポート範囲

DS-CUDA は CUDA C/C++ コンパイラでコンパイル可能なすべてのソースコードを扱えるわけではありません。現在のところ、アプリケーションプログラムのソースコードが以下の条件に該当する記述を含む場合には、そのソースコードは処理できません。

- テクスチャリファレンスおよびデバイス変数の名前空間は無視されます。従って異なる名前空間内に同一名のテクスチャリファレンスやデバイス変数が存在するコードは扱えません。

  扱えない例:

```
namespace foo {
  texture<float, 1, cudaReadModeElementType> MyTex0;
  __constant__ int MyVar0;
  ...
}
namespace bar {
  texture<float, 1, cudaReadModeElementType> MyTex0;
  __constant__ int MyVar0;
  ...
}
```

- CUDA カーネルの名前空間は概ね正しく扱われますが、無名の名前空間は無視されます。従って複数の無名の名前空間内に同一名の CUDA カーネルが存在するコード

は扱えません。

扱えない例:

```
namespace {
  __global__ void myFunc0(void)
  {
      ....
  }
}
namespace {
  __global__ void myFunc0(void)
  {
      ....
  }
}
```

扱える例:

```
namespace foo {
  __global__ void myFunc0(void)
  {
      ....
  }
}
namespace bar {
  __global__ void myFunc0(void)
  {
      ....
  }
}
```

- (従来のバージョンでは扱えなかった、CUDA カーネルを関数ポインタ経由で呼び出す記法は DS-CUDA バージョン 2.0.0 で扱えるようになりました。)

バージョン 2.0.0 で扱えるようになった例:

```
void (*kernel)(...);

__global__ void myKernel(...)
{
  ....
}
...
void main(void)
{
  kernel = myKernel;
  ...
  kernel<<<grid, threads>>>(...);
}
```

5.3 通信インタフェース

InfiniBand Verb インタフェース : ホストと GPU デバイスが InfiniBand を介して通信する場合に使用するインタフェースです。OFED の提供する C 言語ライブラリ関数を用いて記述されています。ライブラリの詳細については OFED の提供するドキュメントを参照して下さい。

TCP/IP インタフェース : ホストと GPU デバイスが TCP/IP を介して通信する場合に使用するインタフェースです。UNIX において一般的な BSD ソケット API を用いて記述されています。通信にはポート番号 65432 ∼ 65436 を使用します。

5.4 CUDA カーネルのアドレス取得

クライアントが CUDA カーネルの実行を DS-CUDA サーバへ要求する際、カーネルの同定はカーネル関数の名前を用いて行います。より具体的には、g++ の組み込み関数 __PRETTY_FUNCTION__ を使用してカーネル関数の名前 (引数や C++ テンプレートパラメタの情報を含む) を取得し、この文字列をサーバへ通知します。

サーバは受け取った __PRETTY_FUNCTION__ の値から関数の signature を生成します。生成にはコマンド $dscudapkg/bin/pretty2mangled を用います。次にサーバは自身に対してコマンド /usr/bin/nm を実行し、その signature に対応するアドレスを取得します。このアドレスを CUDA API cudaLaunch() へ与え、対応するカーネルを実行します。

このアドレス取得処理は各カーネルの最初の呼び出し時にだけ実行されます。アドレス取得処理によるオーバヘッドを低減するために、2 度目以降のカーネル呼び出し時には、

1 回目に取得したアドレスが再利用されます。

例:

```
__PRETTY_FUNCTION__    void foobar::vecAddT(T1*, T1*) [with T1 = double]

 ↓  pretty2mangled

関数 signature         _ZN6foobarL7vecAddTIdEv

 ↓ nmfoo.svr | grep _ZN6foobarL7vecAddTIdEv

関数アドレス           0x0000000000adac78

 ↓

cudaLaunch(0x0000000000adac78);
```

6 利用許諾

DS-CUDA ソフトウェアパッケージの利用条件についてはファイル $dscudapkg/00license-j をご確認下さい。

7 DS-CUDA ソフトウェアパッケージ更新履歴

version	date	description
2.5.0	05-Aug-2015	サーバのライブマイグレーション機能を実装。 型番の異なる GPU 同士での冗長デバイス構成に対応。 チェックポイントにおいてデバイスメモリイメージ を保存する機能を実装。
2.4.0	27-Mar-2015	P2P 通信を GPU Direct ver2, ver3 に対応。
2.2.0	24-Feb-2015	新しい API、`dscudaMemcopies()`, `dscudaBroadcast()` を追加。 P2P 通信の実装 (`cudaMemcpy()` の UVA 対応、`cudaMemcpyPeer()` の実装)。
2.1.0	11-Aug-2014	機能強化 (`dscudasvr` の自動生成、自動転送など)
2.0.0	29-Jul-2014	Dynamic Parallelism をサポート。
1.2.9	05-Feb-2013	デーモン `dscudad` を導入。
1.2.3	24-Sep-2012	一般公開向けに利用許諾等を整備。
1.2.1	09-Aug-2012	初版作成。