DE LA RECHERCHE À L'INDUSTRIE

www.cea.fr

DÉTECTION AUTOMATIQUE DU GPU LE PLUS PROCHE DANS MPC

Hugo Taboada, Julien Jaeger

CEA,DAM,DIF, F-91297 Arpajon, France

07/05/2015

SOMMAIRE

- Introduction
  - Objectifs
  - MPC
  - Sélection d'un GPU en CUDA
  - Détection de la topologie (Hwloc)
- Implémentation
  - Détection du GPU le plus proche
  - Notre approche
  - Gestion des contextes CUDA
- Résultats
  - Vérifications CUDA
  - Héritage OpenACC
- Conclusion / Travaux futurs

INTRODUCTION

OBJECTIFS DU STAGE

- **Permettre une sélection automatique de GPU**

- De base tous les threads sont sur le même GPU en CUDA
  - L'utilisateur a la possibilité de spécifier le GPU à utiliser
    - Sans connaissance de la topologie → Répartition inefficace
- Détection du GPU le plus proche
- Intégration à l'ordonnanceur MPC
- **Faciliter l'utilisation de plusieurs GPUs sur un nœud de calcul**
  - Exemple : utilisation de 2 GPUs en MPI sur un nœud de calcul à 24 coeurs

- **Gestion des threads provenant de bibliothèques externes**

- Les bibliothèques externes créent leurs propres threads
- Le but est d'ordonnancer tous ces threads efficacement
  - Cas tests: thread d'asynchronisme
    - **Collectives non-bloquantes**
    - Gestion des threads OpenACC, OpenMP4

PRÉSENTATION DE MPC

- MultiProcessor Computing

- Bibliothèque pour le calcul parallèle.
  - Une implémentation MPI 1.3 à base de thread (thread based)
    - + MPI_THREAD_MULTIPLE, IO, NBC
  - Une implémentation OpenMP 3.1
  - Une implémentation Pthread
- Un allocateur mémoire multithread et topologique
- Compatible gcc, icc
- Logiciel de debug fourni (gdb patché reconnaissant les threads MPC)
- Un logiciel libre (Licence CeCILL-C)
  - La version courante est la 2.5.2

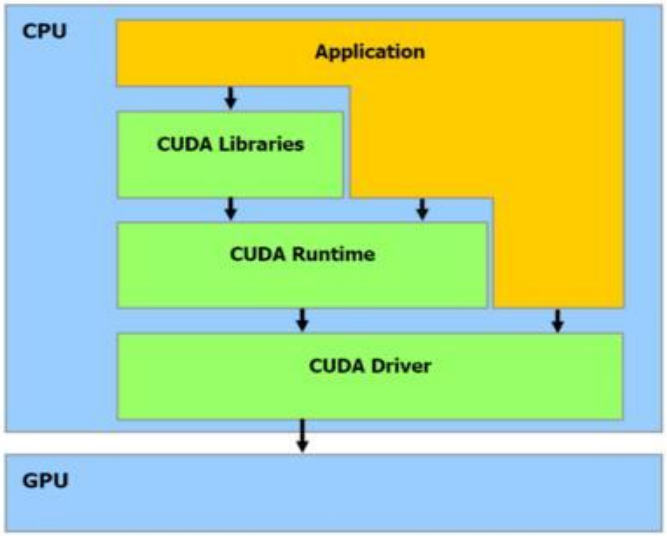
CHOIX DU GPU EN CUDA

- Rappel sur CUDA

- Extension du langage C qui permet la programmation sur les GPGPUs Nvidia
  - Version courante 7.0, Compute capabilities 5.2

- Fonctions de l'API

- Deux API :
  - Runtime : Fonctions usuelles (cudaMalloc, cudaFree, …)
  - Driver : Fonctions de plus bas niveaux (cuCtxCreate, cuCtxPushCurrent, …)
- Comment choisir un GPU ?
  - Utiliser cudaSetDevice(int device) de l'API runtime
  - Utiliser la création de contexte de l'API driver



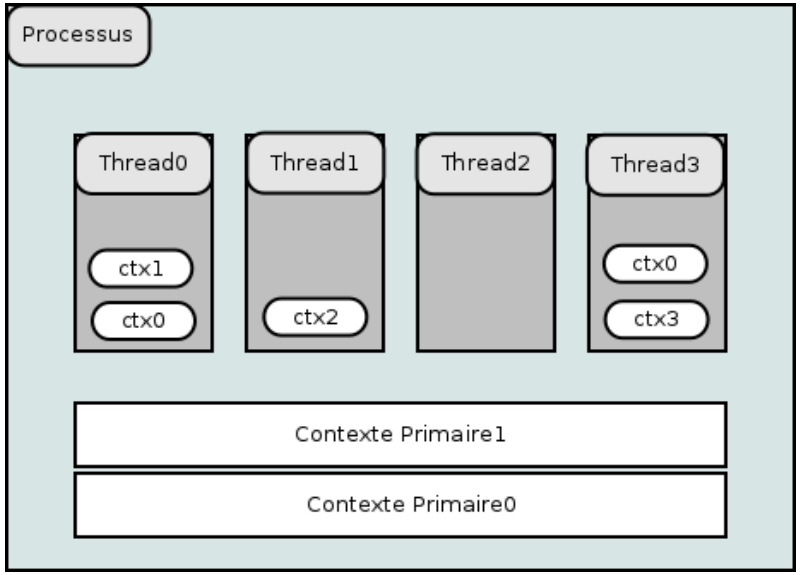
FONCTIONNEMENT INTERNE DES CONTEXTES

- Contexte classique → 1 contexte par thread

- Avant CUDA 4.0
- Chaque thread accède à son propre GPU
- Impossibilité d'avoir 2 threads par GPU en même temps
- Possibilité de les utiliser avec l'API driver

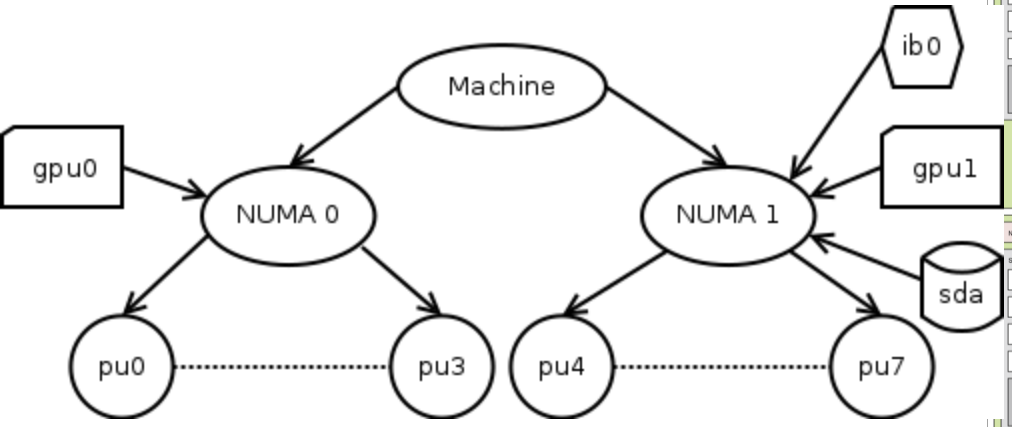
- Contexte primaire → 1 contexte par processus

- Depuis CUDA 4.0
- Tous les threads accède au même GPU de façon concurrente
- Est créé automatiquement lors d'un appel CUDA si un contexte classique n'est pas déjà présent.



HARDWARE LOCALITY (HWLOC)

- Rappel sur Hardware Locality (hwloc)
  - Logiciel permettant de connaître la topologie d'une machine
  - Fusion de deux projets :
    - « Libtopology (équipe Runtime de l'INRIA Bordeaux)
    - « Portable Linux Processor Affinity (PLPA) » (équipe d'OpenMPI)
  - Maintenu par l'équipe de l'INRIA Bordeaux et intégré dans OpenMPI
    - Version stable → 1.10
- Modes de fonctionnements
  - Ligne de commande
    - lstopo



HARDWARE LOCALITY (HWLOC)

- Modes de fonctionnements (suite)

  - Avec l'interface de programmation en C

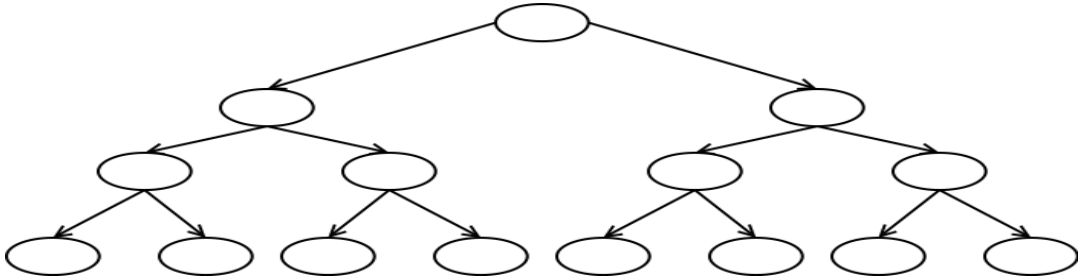
```c
/* Allocate and initialize topology object. */
hwloc_topology_init(&topology);
hwloc_topology_load(topology);

/* Optionally, get some additional topology information
   in case we need the topology depth later. */
topodepth = hwloc_topology_get_depth(topology);

printf("Topology have a depth of %d\n",topodepth);
for (depth = 0; depth < topodepth; depth++) {
    printf("*** Objects at level %d\n", depth);
    for (i = 0; i < hwloc_get_nbobjs_by_depth(topology, depth); i++) {
        hwloc_obj_snprintf(string, sizeof(string), topology,
                hwloc_get_obj_by_depth(topology, depth, i),
                "#", VERBOSE);
        printf("Index %u: %s\n", i, string);
    }
}
```

```
Topology have a depth of 8
*** Objects at level 0
Index 0: Machine#0
*** Objects at level 1
Index 0: NUMANode#0
Index 1: NUMANode#1
*** Objects at level 2
Index 0: Socket#0
Index 1: Socket#1
*** Objects at level 3
Index 0: L3Cache
Index 1: L3Cache
*** Objects at level 4
Index 0: L2Cache
.....
Index 23: L2Cache
*** Objects at level 5
Index 0: L1dCache
.....
Index 23: L1dCache
*** Objects at level 6
Index 0: Core#0
.....
Index 23: Core#13
*** Objects at level 7
Index 0: PU#0
.....
Index 23: PU#23
```

HARDWARE LOCALITY (HWLOC)

- Fonctionnalités déjà intégrées à MPC

- A quoi sert hwloc actuellement dans MPC ?
  - Localité topologique dans le runtime OpenMP
    - Exemple : **#pragma omp barrier** → arbre topologique d'attente



  - Allocateur mémoire topologique
    - Alloue la mémoire en fonction de la topologie(NUMA)
  - Localité des cartes InfiniBand (en cours)

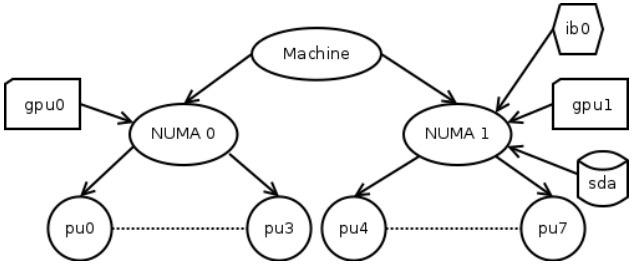
- Version utilisée dans MPC → 1.10

IMPLÉMENTATION

DÉTECTION DU GPU LE PLUS PROCHE

Utilisation de l'API de Hwloc

- Hwloc permet de récupérer les informations concernant les devices (type, pci_bus_id…)
- Nous parcourons la liste des devices détectés par Hwloc
- En faisant un appel de fonction de l'API, nous pouvons savoir à quels objets de la topologie nous sommes rattachés
- En redescendant l'arbre topologique, nous trouvons les CPUs les plus proches



Sauvegarder la topologie

- Création d'une table de correspondance (actuellement)
  - Chaque CPU est associé au device le plus proche
- Création d'une matrice de distance
  - Chaque CPU est associé à tous les devices en prenant en compte leur distance

Interface avec le runtime CUDA

- Fonction de MPC
  - sctk_get_cpu() : utilise l'API de Hwloc pour retourner le CPU sur lequel le thread est lié
- Fonction de l'API driver de CUDA
  - cuDeviceGetByPCIBusId(Cudevice *dev,const char *pciBusId)

NOTRE APPROCHE

- Pas de contextes primaires.

Impossibilité de répartir le travail de plusieurs threads sur plusieurs GPU

- Utilisation des contextes « classiques »

Une pile de contextes par thread
  - Stocker dans la TLS
  - Le sommet de pile correspond au contexte en cours d'utilisation

Obligation d'appeler une fonction API Driver pour créer un contexte « classique »
  - Fonction cuCtxCreate()
  - Depuis CUDA 4.0, ce n'est plus le comportement par défaut
  - Un fois qu'un contexte « classique » est présent, les autres contextes créés par l'API Runtime s'empilent sur celui-ci

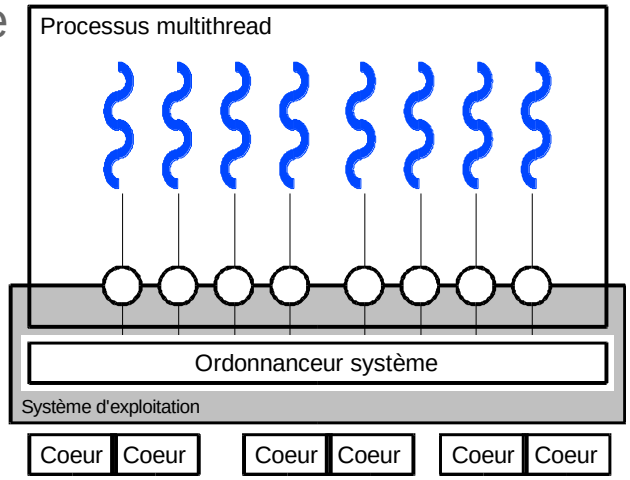
- Difficulté : ordonnanceur MxN dans MPC

ORDONNANCEMENT DE THREADS

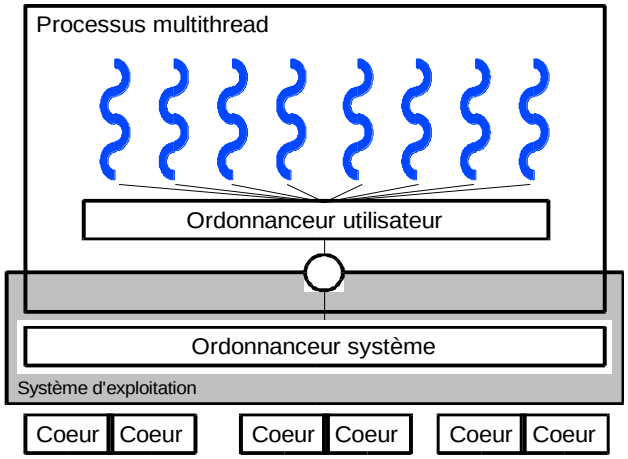
- Fonctionnement général

Il existe trois types de bibliothèques de threads :

- Bibliothèque système



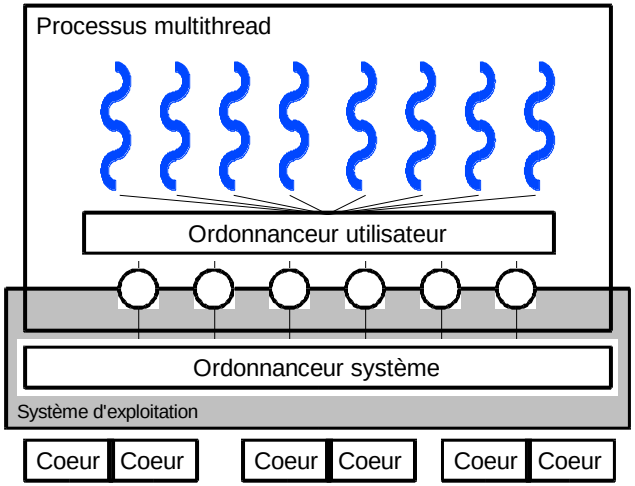
- Bibliothèque utilisateur



ORDONNANCEUR MPC

- Ordonnanceur de MPC

- Bibliothèque mixte (ou MxN)



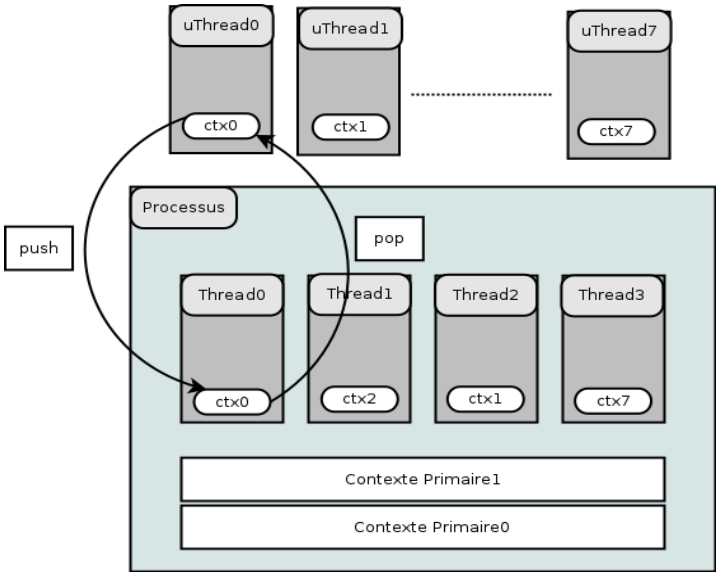
- Fonctionnement actuel
  - Ordonnance équitablement tous les threads (MPI, OpenMP, Pthread)

- Fonctionnement dans le futur
  - Ordonnancer les threads en fonction d'un type de thread (modèle de programmation)
  - Permettre l'ordonnancement de groupe (gang schedulling)

GESTION DES CONTEXTES CUDAS

- Intégration avec l'ordonnanceur de MPC

- Création d'un contexte « classique »
  - cuCtxCreate → à l'initialisation de chaque thread
- Besoin de sauvegarder le contexte CUDA lors de l'ordonnancement de threads:
  - cuCtxPushCurrent → à l'ordonnancement d'un thread
  - cuCtxPopCurrent → lorsqu'un thread est désordonnancé
- Flags fournis à la création pour spécifier la politique d'attente du thread
  - Le flag CU_CTX_SCHED_YIELD permet de rendre la main au CPU lorsque l'on attend le GPU (fin de kernel, transfert mémoire, ….)



EXPÉRIMENTATIONS

VERIFICATION DE VALIDITÉ CODE CUDA

- Précédemment…

- Machine Cirrus
  - 2 GPUs par nœud
  - 12 cœurs par nœud NUMA

- Utilisation de l'API runtime (contexte primaire)
  - 24 cœurs sur 1 GPU
  - 1 cœur sur 2 GPUs

- Utilisation de l'API driver (ordonnanceur système)
  - Gestion des contextes classique à la main → Très fastidieux
    - 24 appels à cuCtxCreate()
  - Détection de la topologie à la main → Très fastidieux

VERIFICATION DE VALIDITÉ CODE CUDA

- Maintenant
  - Machine Cirrus
    - 2 GPUs par nœud
    - 12 cœurs par nœud NUMA

```
int main(int argc, char** argv){
    int rank, size;
    int dev;

    MPI_Init(&argc,&argv);

    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    cudaGetDevice(&dev);

    printf("hello world, rank=%d, size=%d, dev=%d\n",
        rank, size, dev);
    fflush(stdout);

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

```
hello world, rank=2, size=24, dev=1
hello world, rank=18, size=24, dev=1
hello world, rank=14, size=24, dev=0
hello world, rank=22, size=24, dev=0
hello world, rank=3, size=24, dev=1
hello world, rank=0, size=24, dev=1
hello world, rank=23, size=24, dev=0
hello world, rank=13, size=24, dev=0
hello world, rank=6, size=24, dev=1
hello world, rank=7, size=24, dev=1
hello world, rank=4, size=24, dev=1
hello world, rank=1, size=24, dev=0
hello world, rank=12, size=24, dev=0
hello world, rank=16, size=24, dev=0
hello world, rank=11, size=24, dev=1
hello world, rank=20, size=24, dev=0
hello world, rank=19, size=24, dev=0
hello world, rank=9, size=24, dev=1
hello world, rank=17, size=24, dev=0
hello world, rank=5, size=24, dev=1
hello world, rank=21, size=24, dev=0
hello world, rank=15, size=24, dev=0
hello world, rank=10, size=24, dev=1
hello world, rank=8, size=24, dev=1
```

HERITAGE OPENACC

- Principe

- L'implémentation PGI du runtime OpenACC hérite des contextes CUDAs.
  - Simplification du code dans OpenACC
  - Choix du GPUs optimal sans intervention de l'utilisateur

- Notre travail se répercute sur OpenACC (PGI compilers)

CONCLUSION

CONCLUSION

- Conclusion sur le travail

- Intégration de la détection des GPUs les plus proches dans MPC
  - Hwloc nous permet de détecter la topologie
  - Utilisation des contextes « classiques » pour répartir les threads sur les GPUs
  - Le flag SCHED_YIELD permet de rendre la main à l'ordonnanceur lorsqu'on attend le GPU

- Des premiers tests de fonctionnement ont été effectués:
  - Gestion des contextes lors de l'ordonnancement des threads MPC
  - Premiers tests multi-GPUs fonctionnels

- Quelques limitations dues aux contextes « classiques »
  - Plusieurs threads ne peuvent pas utiliser le même GPU en même temps.

TRAVAUX FUTURS

- Travaux futurs sur la gestion des GPUs

- Utiliser des eTLS pour les GPUs
  - eTLS d'origine: une tls par tâche MPI de MPC
  - Chaque GPU aurait sur l'hôte sa propre zone privée partagée entre plusieurs threads
  - Permettrait de partager un GPU entre plusieurs (pas tous) threads

- Utilisation du GPU le plus proche en permanence (dans un prochain stage)
  - Annotation de codes pour signifier que le thread peut redéfinir son GPU le plus proche
  - Traquer tous les cudaMalloc() et les cudaFree()

- Tests de fonctionnalités restant:
  - Valider la détection des GPUs
  - Valider l'héritage des contextes en OpenACC

TRAVAUX FUTURS

- Futurs travaux de stage

- Gestion des threads connus par l'ordonnanceur
  - La libNBC a été intégré à MPC
  - Les threads de progression sont des threads MPC
  - Gestion des threads de différentes natures
    - **Placement par rapport aux autres threads**
    - Politique d'ordonnancement spécialisée

- Futurs travaux de thèse

- Comment reconnaitre et ordonnancer les threads spécialisés de bibliothèques externes
  - Première approche sur les threads de progression libNBC

Commissariat à l’énergie atomique et aux énergies alternatives
Centre DAM-Ile de France- Bruyères-le-Châtel 91297 Arpajon Cedex
T. +33 (0)1 69 26 40 00 | FAX +33 (0)1 69 26 40 00

Etablissement public à caractère industriel et commercial | RCS Paris B 775 685 019