CUDA 4 et architecture Fermi
GPUDirect2.0 et UVA

Pierre Kunzli, Paul Albuquerque - hepia

Projet TeraFlo, rapport intermédiaire, 21 mars 2012

1 Introduction

La version 4.0 de CUDA introduit avec l'architecture Fermi deux nouveautés concernant la gestion de la mémoire sur les devices.

1. *GPUDirect2.0*, qui permet d'échanger des données entre deux devices (fonctionnalité appelée *peer-to-peer* fig.1);
2. l'*Unified Virtual Addressing* (fig.2), qui simplifie la programmation, du fait qu'un espace d'adressage unifié est utilisé pour le host et les devices.

De plus, la gestion de plusieurs devices est maintenant facilitée.

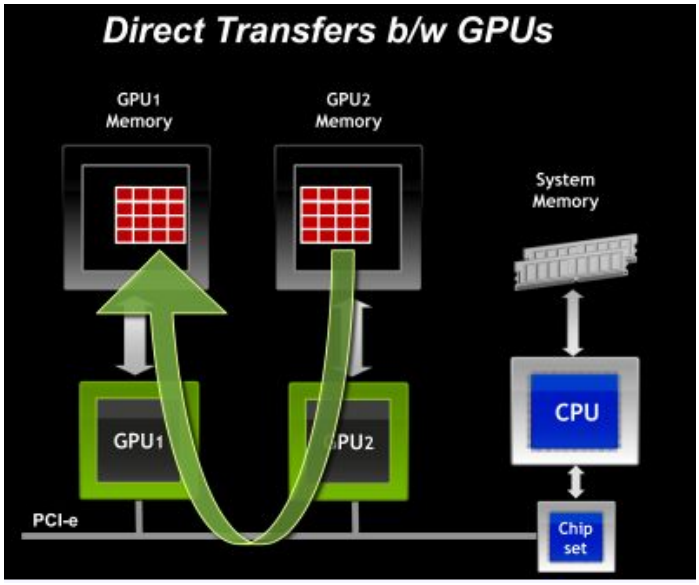

Figure 1: Echange peer-to-peer entre deux devices

2 Pré-requis

Afin de profiter de ces fonctionnalités, il faut que le runtime CUDA installé soit en version ≥ 4.0 et les devices doivent avoir un *CUDA Capability Major version* ≥ 2 (architecture Fermi). Afin de pouvoir utiliser l'UVA, l'application doit être compilée en 64 bits et le mode *TCC* (*Tesla Compute Cluster*) doit être activé si l'application fonctionne sur Windows 7/Vista (voir [1] section 3.6).

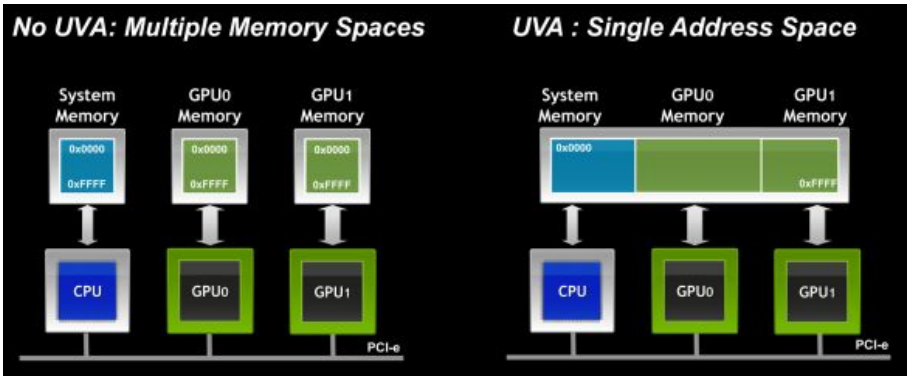

Figure 2: Unified Virtual Addressing

3 Mise en oeuvre

3.1 Contrôle des pré-requis

Le code suivant permet de s'assurer que le programme s'exécute sur une machine supportant les fonctionnalités UVA et P2P :

```
#define checkCudaErrors(err)           __checkCudaErrors (err, __FILE__, __LINE__)
inline void __checkCudaErrors( cudaError err, const char *file, const int line ){
    if( cudaSuccess != err) {
        fprintf(stderr, "%s(%i) : CUDA Runtime API error %d: %s.\n",
                file, line, (int)err, cudaGetErrorString( err ) );
        exit(-1);
    }
}
inline bool IsAppBuiltAs64(){
    #if defined(__x86_64) || defined(AMD64) || defined(_M_AMD64)
        return 1;
    #else
        return 0;
    #endif
}
inline bool IsCuda4(){
    #if CUDART_VERSION >= 4000
        return 1;
    #else
        return 0;
    #endif
}
// controler que l'application est compilee en 64 bits
if (!IsAppBuiltAs64()) {
    cout<<"UVA is only supported on 64-bit OSs"
       <<"the application must be built as a 64-bit target"<<endl;
    exit(-1);
}
// controler que cuda4 est disponible
if (!IsCuda4()) {
    cout<<"UVA needs cuda4 runtime"<<endl;
    exit(-1);
}
```

```
// controler que chaque device supporte p2p et uva
int numGpus;
checkCudaErrors(cudaGetDeviceCount(&numGpus));
cudaDeviceProp deviceProp;
for(int i=0;i<numGpus;i++){
    checkCudaErrors(cudaGetDeviceProperties(&deviceProp, i));
    if(!(deviceProp.major>=2)){
        cout<<"GPU 2.0 (fermi) requied to use UVA/P2P"<<endl;
        exit(-1);
    }
}
```

3.2 UVA

Lorsque les conditions sont remplies, l'UVA est automatiquement activé. Un seul espace d'adressage est alors utilisé pour le host et pour les devices de *compute capability* ≥ 2. Cet espace d'adressage est utilisé pour toute allocation faite sur les devices, mais uniquement pour les allocations avec `cudaHostAlloc()` ou `cudaMallocHost()` sur la mémoire du host.
La première conséquence est que, lors des `cudaMemcpy()` sur ou depuis n'importe quel device qui emploie l'UVA, on peut utiliser la valeur `cudaMemcpyDefault` pour spécifier le type de copie. Le runtime est capable de déterminer la source et la destination à partir des pointeurs.
Ensuite, lorsque l'UVA est utilisé, les pointeurs alloués avec `cudaHostAlloc()` ou `cudaMallocHost()` peuvent être directement déréférencés depuis les kernels, et donc, la mémoire du host directement accédée. Il faut cependant utiliser cette fonctionalité avec précaution du fait des forts temps de latence qu'elle implique.

```
__global__ void kernel(int* ptr)
{
    int i = ptr[0];
    ptr[1] = 3;
}
...
// allouer de la memoire sur le host
int* hostPtr1;
checkCudaErrors(cudaMallocHost(&hostPtr1,10*sizeof(int)));
int* hostPtr2 = (int*)malloc(10*sizeof(int));
...
// la memoire allouee avec cudaMallocHost est directement accessible depuis le kernel
kernel<<<dimGrid, dimBlock>>>(hostPtr1);
...
// allouer de la memoire sur le device 0
checkCudaErrors(cudaSetDevice(0));
int* devicePtr;
checkCudaErrors(cudaMalloc((void**)&devicePtr,10*sizeof(int)));
// copier des donnees du host vers le device, inutile de specifier le type de copie
// le runtime determine automatiquement la source et la destination
checkCudaErrors(cudaMemcpy(hostPtr2,devicePtr,10*sizeof(int),cudaMemcpyDefault));
```

3.3 Peer-to-Peer

GPUDirect 2 permet des échanges directs entre deux devices, appelés copies *peer-to-peer*. Il est possible d'utiliser cette fonctionnalité sans que l'UVA soit actif. Cependant, cela implique que les données transitent par le host et cette façon de faire ne permet donc pas de gain de performance. Dans ce cas, il faut faire usage des fonctions de copies standard avec le suffixe *Peer*. Par exemple, `cudaMemcpyPeer()`,

qui prend en paramètre le pointeur destination, le device destination, le pointeur source, le device source, et enfin la taille des données à copier.

```
// allouer de la memoire sur le device 0
checkCudaErrors(cudaSetDevice(0));
int* device0Ptr;
checkCudaErrors(cudaMalloc((void**)&device0Ptr,10*sizeof(int)));
// allouer de la memoire sur le device 1
checkCudaErrors(cudaSetDevice(1));
int* device1Ptr;
checkCudaErrors(cudaMalloc((void**)&device1Ptr,10*sizeof(int)));
// copier les donnees du device 0 sur le device 1
checkCudaErrors(cudaMemcpyPeer(device1Ptr, 1, device0Ptr, 0, 10*sizeof(int)));
```

3.4 Peer-to-Peer avec UVA

Lorsque l'UVA est activé, un device peut accéder directement à la mémoire d'un autre device sans passer par le host. L'accès P2P doit être activé entre chaque device concerné. Une fois le P2P activé, la mise en oeuvre est très simple puisque qu'il suffit d'utiliser la fonction `cudaMemcpy()`, le runtime étant capable de déterminer la source et la destination à partir des pointeurs grâce à UVA. Ce type de copie ne transitant pas par le host, les performances sont bien meilleures que sans UVA. A titre d'exemple, l'échange de 500Mo de données entre deux devices prends 346ms en passant par le host et 144ms en copie directe entre devices.

```
// allouer de la memoire sur le device 0
checkCudaErrors(cudaSetDevice(0));
int* device0Ptr;
checkCudaErrors(cudaMalloc((void**)&device0Ptr,10*sizeof(int)));
// activer acces p2p depuis device 0 sur device 1
checkCudaErrors(cudaDeviceEnablePeerAccess(1, 0));
// allouer de la memoire sur le device 1
checkCudaErrors(cudaSetDevice(1));
int* device1Ptr;
checkCudaErrors(cudaMalloc((void**)&device1Ptr,10*sizeof(int)));
// activer acces p2p depuis device 1 sur device 0
checkCudaErrors(cudaDeviceEnablePeerAccess(0, 0));
checkCudaErrors(cudaMemcpy(device1Ptr,device0Ptr,10*sizeof(int),cudaMemcpyDefault));
```

De plus, lorsque l'UVA et l'accès P2P sont activés, un kernel peut accéder directement à la mémoire d'un autre device. Là encore, il faut faire attention aux temps de latence lors de tels accès.

```
__global__ void kernel(int* ptr)
{
    int i = ptr[0];
    ptr[1] = 3;
}
...
// allouer de la memoire sur le device 0
checkCudaErrors(cudaSetDevice(0));
int* device0Ptr;
checkCudaErrors(cudaMalloc((void**)&device0Ptr,10*sizeof(int)));
// activer acces p2p depuis device 0 sur device 1
checkCudaErrors(cudaDeviceEnablePeerAccess(1, 0));
// allouer de la memoire sur le device 1
checkCudaErrors(cudaSetDevice(1));
int* device1Ptr;
```

```
checkCudaErrors(cudaMalloc((void**)&device1Ptr,10*sizeof(int)));
// activer acces p2p depuis device 1 sur device 0
checkCudaErrors(cudaDeviceEnablePeerAccess(0, 0));
// le kernel va s'executer sur le device 1 et acceder directement a la memoire du device 2
kernel<<<dimGrid, dimBlock>>>(device0Ptr);
...
```

4 Utilisation de plusieurs devices

Comme on peut le voir dans les exemples, un seul processus sur le CPU est capable de prendre le contôle de plusieurs devices. En effet, il est maintenant possible de lancer des exécutions asynchrones à partir d'un thread, puis de prendre le contrôle d'un autre device afin de lancer d'autres exécutions. Cela facilite dans certains cas la gestion de la programmation multi-devices puisqu'il n'est plus nécessaire d'utiliser plusieurs threads ou processus CPU.
Dans l'autre sens, il est également maintenant possible pour plusieurs threads ou processus de prendre le contrôle d'un même device, les kernels sont alors exécutés en concurrence sur le device concerné. Cela permet de faire fonctionner plusieurs programmes CUDA simultanément ou de simplifier la conception d'un programme dans certains cas.

References

[1] NVidia. Cuda c programming guide 4.1 sections 3.2.6 et 3.2.7, 2012. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[2] Tim C. Schroeder. Peer-to-peer & unified virtual addressing cuda webinar, 2011. http://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf.

A Exemple de code complet

```
#include <cuda_runtime.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

/* fonctions d'aide pour cuda */
#define getLastCudaError(msg)   __getLastCudaError (msg, __FILE__, __LINE__)
inline void __getLastCudaError( const char *errorMessage, const char *file, const int line
     )
{
    cudaError_t err = cudaGetLastError();
    if( cudaSuccess != err) {
        fprintf(stderr, "%s(%i) : getLastCudaError() CUDA error : %s : (%d) %s.\n",
                file, line, errorMessage, (int)err, cudaGetErrorString( err ) );
        exit(-1);
    }
}

#define checkCudaErrors(err)        __checkCudaErrors (err, __FILE__, __LINE__)
inline void __checkCudaErrors( cudaError err, const char *file, const int line ){
    if( cudaSuccess != err) {
        fprintf(stderr, "%s(%i) : CUDA Runtime API error %d: %s.\n",
                file, line, (int)err, cudaGetErrorString( err ) );
        exit(-1);
    }
}

inline bool IsAppBuiltAs64(){
#if defined(__x86_64) || defined(AMD64) || defined(_M_AMD64)
    return 1;
#else
    return 0;
#endif
}

inline bool IsCuda4(){
#if CUDART_VERSION >= 4000
    return 1;
#else
    return 0;
#endif
}

// nombre de threads par block
#define NT 128
```

```
// kernel basique
__global__ void kernel(int *d2, int *d1, int *h1, int size)
{
    int myPos = blockIdx.x*NT + threadIdx.x;
    if(myPos<size) d2[myPos]=d1[myPos];
    h1[size-1]=2;
}

int main( int argc, char** argv)
{
    // controler que le systeme possede 2 GPU au moins
    int numGpus;
    checkCudaErrors(cudaGetDeviceCount(&numGpus));
    if(numGpus<2){
        cout<<"this␣application␣needs␣at␣least␣2␣devices"<<endl;
        exit(-1);
    }

    // controler que l'application est compilee en 64 bits
    if (!IsAppBuiltAs64()) {
        cout<<"UVA␣is␣only␣supported␣on␣64-bit␣OSs␣and␣the␣application␣must␣be␣built␣as␣a␣
            64-bit␣target"<<endl;
        exit(-1);
    }
    // controler que cuda4 est disponible
    if (!IsCuda4()) {
        cout<<"UVA␣needs␣cuda4␣runtime"<<endl;
        exit(-1);
    }
    // controler que les deux premiers devices supporte p2p et uva
    cudaDeviceProp deviceProp;
    for(int i=0;i<2;i++){
        checkCudaErrors(cudaGetDeviceProperties(&deviceProp, i));
        if(!(deviceProp.major>=2)){
            cout<<"GPU␣2.0␣(fermi)␣requied␣to␣use␣UVA/P2P"<<endl;
            exit(-1);
        }
    }

    int size = 200;
    // prendre le controle du device 0
    checkCudaErrors(cudaSetDevice(0));
    // activer l'acces p2p du device 0 sur le device 1
    checkCudaErrors(cudaDeviceEnablePeerAccess(1, 0));
    // allouer de la memoire sur le device 0
    int *device0Ptr;
    checkCudaErrors(cudaMalloc((void**)&device0Ptr,size*sizeof(int)));

    // prendre le controle du device 1
    checkCudaErrors(cudaSetDevice(1));
    // activer l'acces p2p du device 1 sur le device 0
    checkCudaErrors(cudaDeviceEnablePeerAccess(0, 0));
    // allouer de la memoire sur le device 1
    int *device1Ptr;
    checkCudaErrors(cudaMalloc((void**)&device1Ptr,size*sizeof(int)));
```

```
    // allouer de la memoire sur le host accessible depuis les devices
    int *hostPtr;
    checkCudaErrors(cudaMallocHost(&hostPtr,size*sizeof(int)));

    hostPtr[size-1]=0;
    // affiche 0
    cout<<hostPtr[size-1]<<endl;

    // effectuer une copie du device 0 sur le device 1
    // (automatiquement p2p, pas besoin de specifier le type de copie)
    checkCudaErrors(cudaMemcpy(device1Ptr,device0Ptr,size*sizeof(int),cudaMemcpyDefault));

    dim3 dimGrid ((size + NT - 1)/NT,1);
    dim3 dimBlock (NT,1);
    checkCudaErrors(cudaSetDevice(0));
    // executer le kernel sur le device 0
    kernel<<<dimGrid, dimBlock>>>(device1Ptr, device0Ptr, hostPtr, size);
    getLastCudaError("copy() execution failed.\n");
    checkCudaErrors(cudaThreadSynchronize());
    // affiche 2 car la memoire du host a ete modifiee par le kernel
    cout<<hostPtr[size-1]<<endl;
}
```