Atelier T7.A1 : Calcul parallèle hybride avec OpenMP, MPI et OpenCL : volet OpenCL

P.F. Lavallée [1], A. Sartirana [2], G. Grasseau [2]

[1] IDRIS, Orsay
email{lavallee}@idris.fr

[2] Laboratoire Leprince-Ringuet, École polytechnique, Palaiseau
email{sartirana, grasseau}@llr.in2p3.fr

JDEV 2013, Palaiseau, 4-6 septembre 2013

Contexte :

- tutoriel . . .
- présentation d' une façon d'aggréger la puissance d'un accélérateur sur le code `hydro` hybride `MPI/OpenMP`
- les principales étapes de notre réflexion
- le développement pose un certains nombres de problèmes
- principes importants de `OpenCL` que l'on découvrira au fur et à mesure.

Transmettre notre expérience sur le cas `hydro`

1. Introduction
   - Motivation
   - Paradigme de programmation
2. `OpenCL`- généralités
3. Organisation `Hydro` hybride
   - Modèle mémoire, modèle d'exécution
   - Gestion de la mémoire avec l'accélérateur
4. Implémentation hybride `MPI/OpenMP/OpenCL`
   - Modèle d'exécution (kernels)
   - Initialisation d'`OpenCL`
5. Conclusion
   - Performances

1 Introduction
- Motivation
- Paradigme de programmation

2 OpenCL- généralités

3 Organisation Hydro hybride
- Modèle mémoire, modèle d'exécution
- Gestion de la mémoire avec l'accélérateur

4 Implémentation hybride MPI/OpenMP/OpenCL
- Modèle d'exécution (kernels)
- Initialisation d'OpenCL

5 Conclusion
- Performances

Pourquoi faire de la programmation hybride sur 3 niveaux ?

	GFlops	*Watts*	*Dollars*
NVidia M2090 ($16 \times 32 = 512$)	666	225	$\simeq$ 2200
Intel E5-2670 (2×8 *coeurs*)	332	230	3100
NVidia K20 ($13 \times 192 = 2496$)	1170	225	$\simeq$ 3000
Intel Ivy Bridge (2×12 *coeurs*)	500 (?)	(?)	(?)

TABLE: Comparaison des performances (double précision), coût, énergie consommée entre processeurs et GPGPU de même génération

Agrégation de la puissance de calculs :

- plusieurs noeuds (MPI)
- plusieurs coeurs (OpenMP, Pthreads, ...)
- pourquoi pas les accélérateurs, GPGPU, coprocesseurs, ..., technologies *many-core* (CUDA, OpenCL, OpenMP 4, OpenACC, ...)

→ GPU intégrés : Intel IvyBridge (HD 4000), AMD Fusion (HD 7000)

Motivation *GFLOPS/Euro* et *GFLOPS/Watt* (coût d'exploitation)

Quel paradigme de programmation ?

Constat, diversité des accélérateurs :

- `NVidia` (`K20, K10, ..., Titan,`),
- `Intel` (`Xeon Phi`),
- `AMD` (`Radeon` et `FirePro` séries, `PS4`, ...),
- ... systèmes embarqués ... `ARM` (`Cortex` séries), `Kalray`, ...

Technologies hybrides ou mixtes :

- `NVidia` (`Logan`)
- `Intel` (`Ivy Bridge`)
- `AMD` (`Fusion`))

Demain, ...

Développements portables, gérant des *hardware* hétérogènes

Introduction

Choix paradigme de programmation (2)

Plusieurs paradigmes de programmation possibles :

- CUDA (directives) très répandu, limitant (uniquement sur cartes NVidia), efficace, nombreux outils. Autre paradigme pour utiliser les processeurs ou d'autres *hardware*
- OpenACC (directives) standard faible, soutenu par NVidia (PGI), pas soutenu par Intel (payant).
- OpenMP 4 (directives) standard mais pas d'implémentation pour l'instant (on en parle ... il faut attendre)
- Intel (directives, API) TBB, Cilk spécifique pour Xeon Phi
- OpenCL (API) standard des systèmes embarques (portables, tablettes, ...) spécifications pour gérer des matériels hétérogènes - peu d'outils (sauf AMD, Intel payant, ...). Utilisé comme intermédiaire par certaines implémentations par directives (portabilité, hétérogène)

Choix d'OpenCL, bas niveau, mais . . . (OpenMP ← Pthreads)

1 Introduction
- Motivation
- Paradigme de programmation

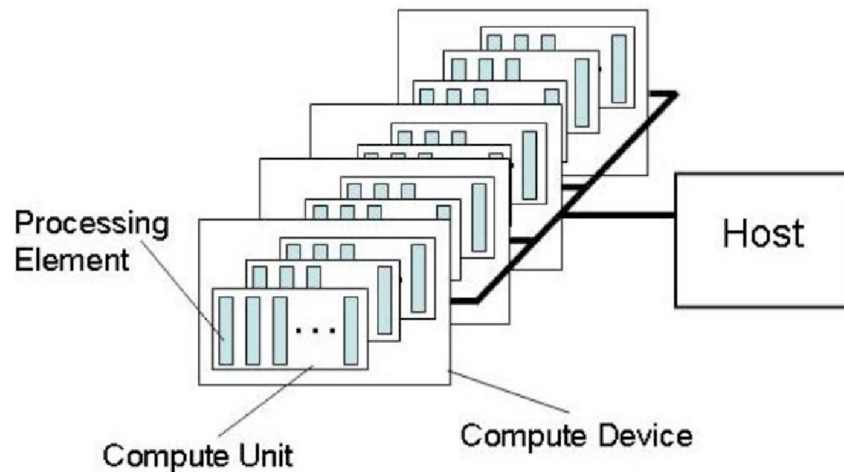
2 OpenCL- généralités

3 Organisation Hydro hybride
- Modèle mémoire, modèle d'exécution
- Gestion de la mémoire avec l'accélérateur

4 Implémentation hybride MPI/OpenMP/OpenCL
- Modèle d'exécution (kernels)
- Initialisation d'OpenCL

5 Conclusion
- Performances

OpenCL = Open **Computing** Language

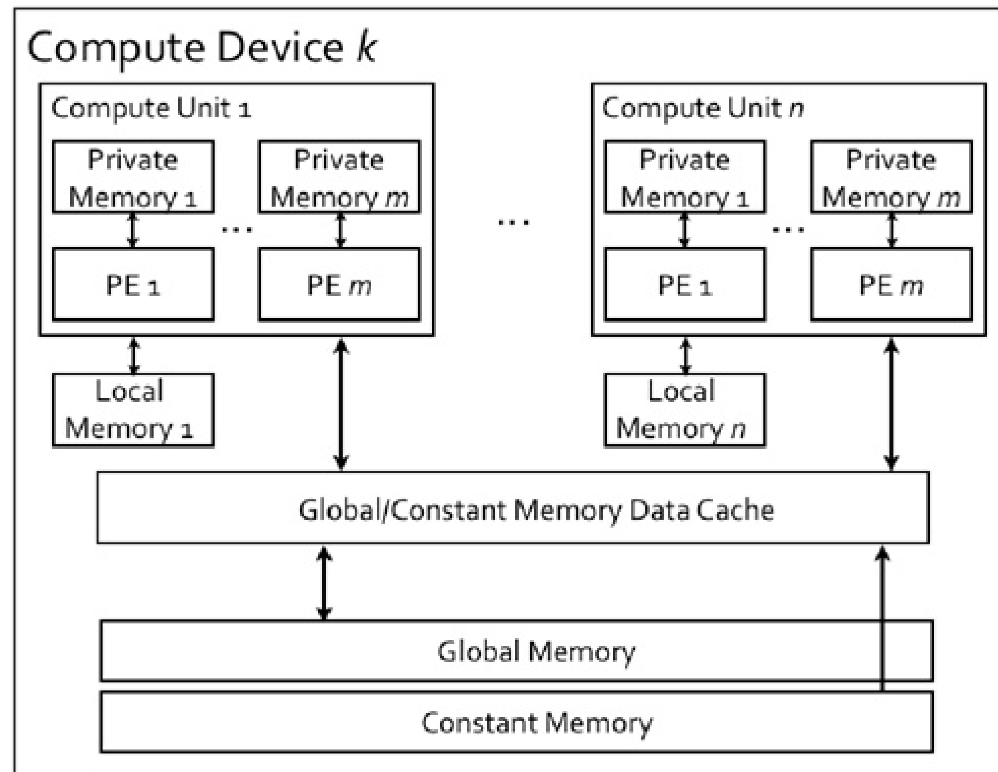


Modèle abstrait ⇒ implémentation sur tout type de *hardware*

- Machine hôte → plusieurs plateformes hétérogènes
- Une plateforme → plusieurs *devices* homogènes
- Un *Device* → plusieurs *Processing Units* (processeurs)
- Un processeur → *Processing Elements*

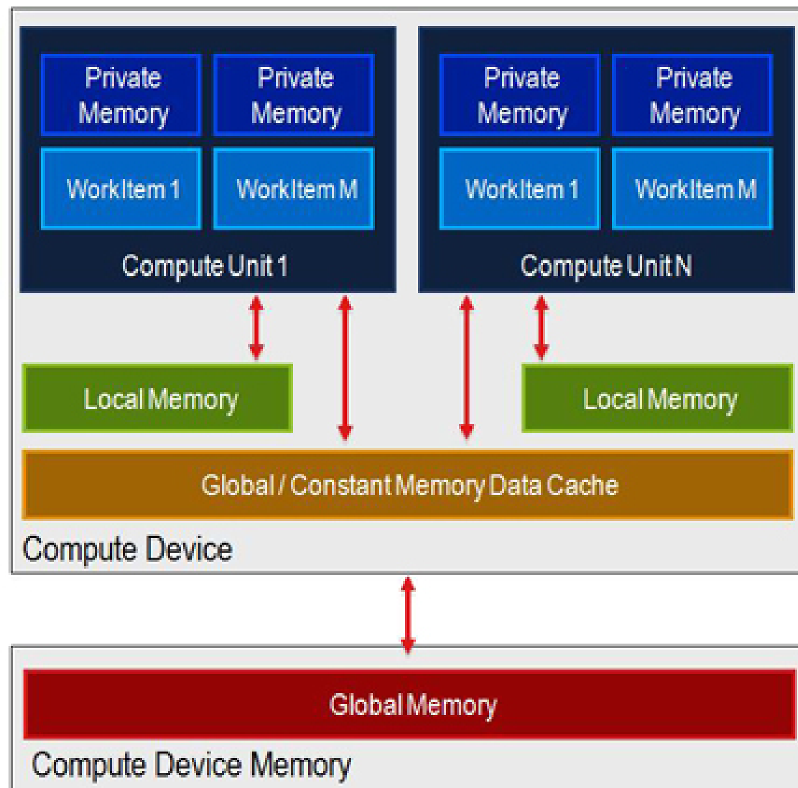
Le modèle représente les caractéristiques des processeurs, accélérateurs, . . . du HPC : les coeurs, les unités vectorielles

Modèle mémoire



- *Global Memory, Constant memory* : partagée par tous les PE/PU
- *Local Memory* : partagée par les PE d'un *Work-group*
- *Private memory* : propre à chaque PE

Exemple d'implémentation sur CPU



Implémentation, potentiellement, très efficace sur CPU

Modèle d'exécution :

- *Host* : serveur de calcul
- *Platform* : ensemble de processeurs de la machine hôte
- *Device* : les processeurs du serveur
- *Processing Units* : un coeur
- *Processing Elements* : unités de traitement vectorielles (AVX, SSE, ...)

Modèle mémoire :

- *Global Memory, Constant memory* : mémoire partagées
- *Local Memory* : registres, cache (volatile)
- *Private memory* : registres, registres AVX, SSE

1 Introduction
- Motivation
- Paradigme de programmation

2 OpenCL- généralités

3 Organisation Hydro hybride
- Modèle mémoire, modèle d'exécution
- Gestion de la mémoire avec l'accélérateur

4 Implémentation hybride MPI/OpenMP/OpenCL
- Modèle d'exécution (kernels)
- Initialisation d'OpenCL

5 Conclusion
- Performances

Organisation `Hydro` hybride

Mémoire

Remarques, évidences sur les modèles de la mémoire :

- `MPI` : mémoire distribuée, échanges de messages entre plusieurs noeuds
- `OpenMP` : mémoire partagée (pas de messages)
- `OpenCL` : extension de la mémoire de l'hôte, plutôt mémoire distribuée (messages), bientôt en mémoire partagée (`Ivy Bridge`,. . .)

Conséquences :

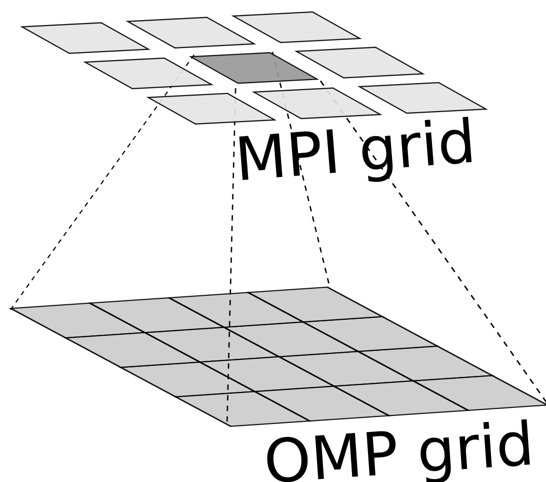
- minimiser les échanges entre l'hôte et la carte accélératrice
- éviter la programmation type "accélérateur" sur des boucles locales, favorisée avec les directives (`OpenMP`, . . .) exécution d'une boucle coûteuse sur le `GPU` (communication/synchronisation).

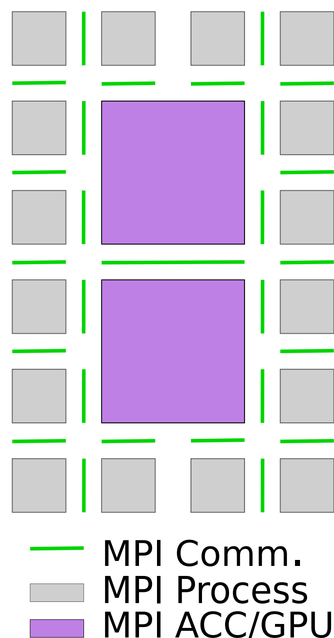
Modèle mémoire : distribuer complètement un domaine sur l'accélérateur
Modèle d'exécution : minimiser le nombre de *kernels*

Kernel : source `C99` avec des extension `OpenCL` (attributs+*built-in*) qui sera compile sur la machine hôte et télé-chargé sur l'accélérateur.

Organisation Hydro hybride

Distribution du travail





Modèles possibles (placement des API) :
MPI → mémoire partagée "au-dessus", mais en ce qui concerne le choix OpenMP/OpenCL ...

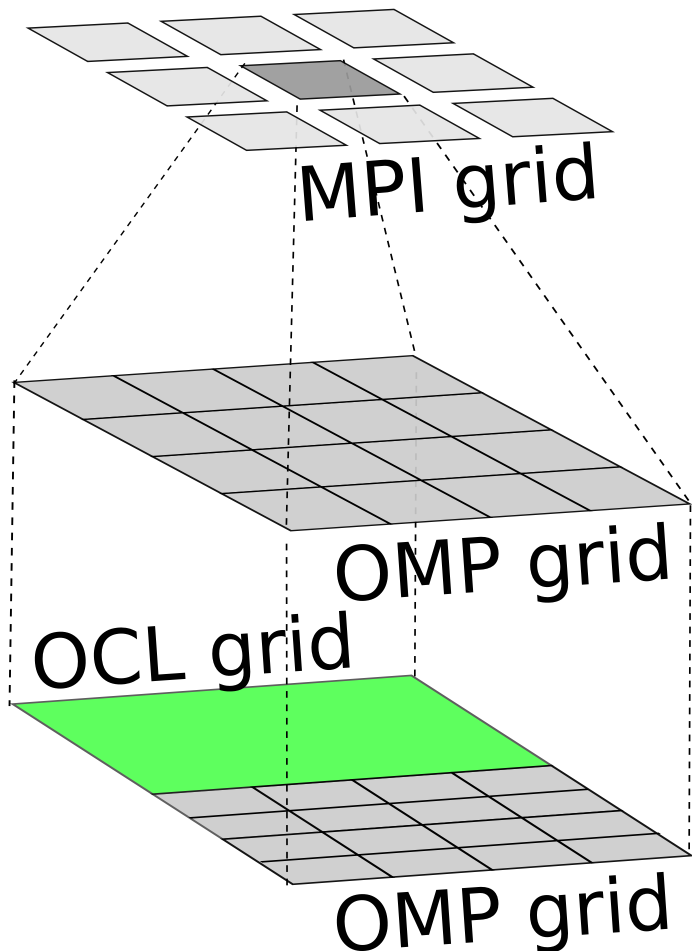
Possibilités :

- MPI/OpenCL possible mais ...
- MPI/OpenMP/OpenCL induit une approche locale "directives" → pas conseillée (vers. *fine grain*).
- (MPI/OpenCL + MPI/OpenMP)
  - un domaine MPI → Accélérateur
  - simple mais gestion des topologies des communicateurs (MPI_CART_CREATE) délicate.
- MPI/(OpenCL + OpenMP)
  - un domaine MPI → 2 domaines : coeurs CPUs, accélérateur

Organisation `Hydro` hybride

Distribution du travail

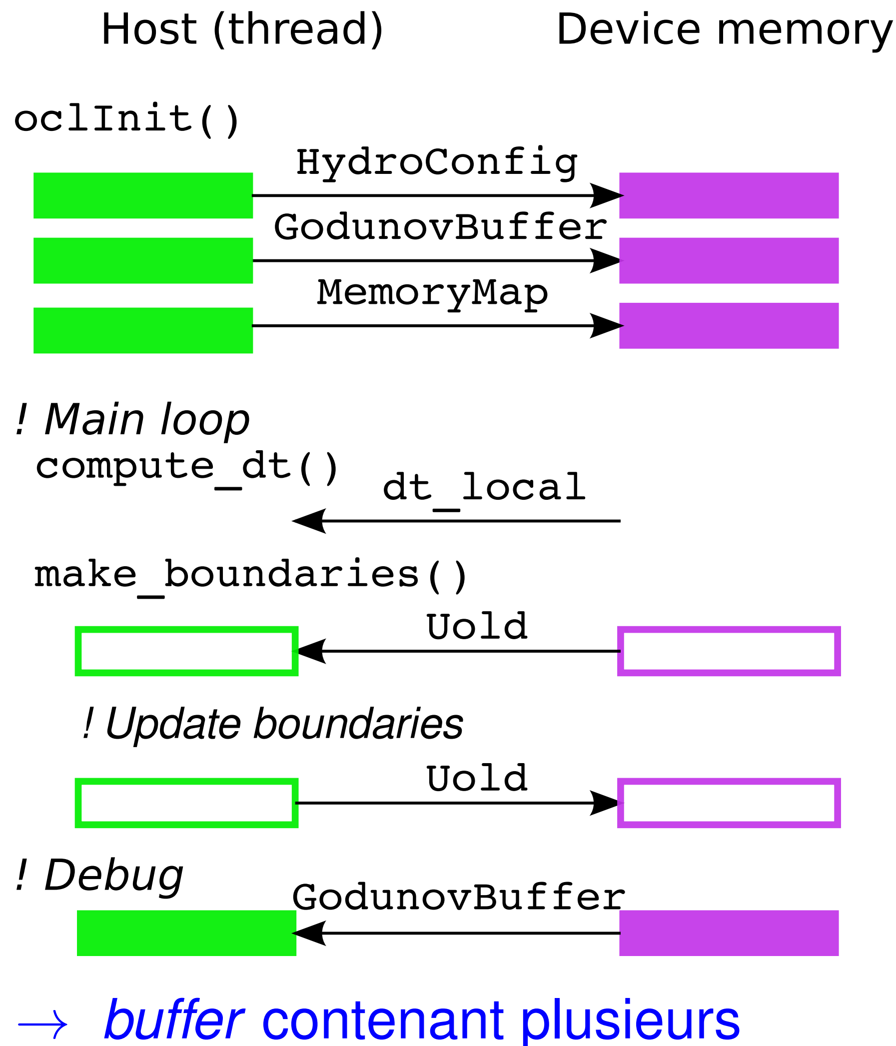
MPI/(OpenCL + OpenMP) → 2 domaines : coeurs CPUs, accélérateur



- Pas de traitement par ligne/colonne (pas de blocage) → problème d'équilibrage de charge (ex : $x = 100 \rightarrow 512$ *streams*)
- Simplification : distribution entre les coeurs et l'accélérateur suivant l'axe des y :
  - $domaine_{cpu} = \{x; y_{min_cpu}, \ldots, y_{max_cpu}\} \rightarrow$ coeurs des CPUs
  - $domaine_{cpu} \rightarrow$ `$OMP_NUM_THREADS`
  - $domaine_{gpu} = \{x; y_{min_gpu}, \ldots, y_{max_gpu}\} \rightarrow$ accélérateur
  - Tout le sous-domaine est sous-traité par l'accélérateur
  - Pas de difficulté technique à généraliser avec une distribution en x et y

Organisation `Hydro` hybride

Gestion de la mémoire



→ *buffer* contenant plusieurs variables

Complications, gestion de la mémoire :

- Pas d'allocation dynamique dans un *kernel* (`malloc` → coûteux)
- Allocation dynamique à partir de l'hôte. Idem en `CUDA`, simplifié par un attribut :

```
__device__ double UoldAcc[N];
```

Autres remarques rendant pénible l'implémentation et favorisant les erreurs :

- morcellement des allocations, leur localisation (hôte, accélérateur)
- arguments dans les fonctions peuvent nombreux (voir `riemann`)
- statut de la mémoire de l'accélérateur (débogage)
- suivit de la mémoire disponible sur l'accélérateur (mémoire 5 − 10*Go*)

Gestion de la mémoire

- 4 principaux *buffers* : `Uold, hydroConfig, hydroMemMap, godunovBuffer`
- nom des *buffers* `xxxxHost, xxxxDevice`
- duplication possible (débogage) : `hydroMemMapHost` ↔ `hydroMemMapDevice`

Configuration

```
typedef struct hydro_config_s{
  // Time control
  real_t t;
  ...
  // Physics
  ...
  // Numerical scheme
  long niter_riemann;
  ...
} HydroConfig_t;
```

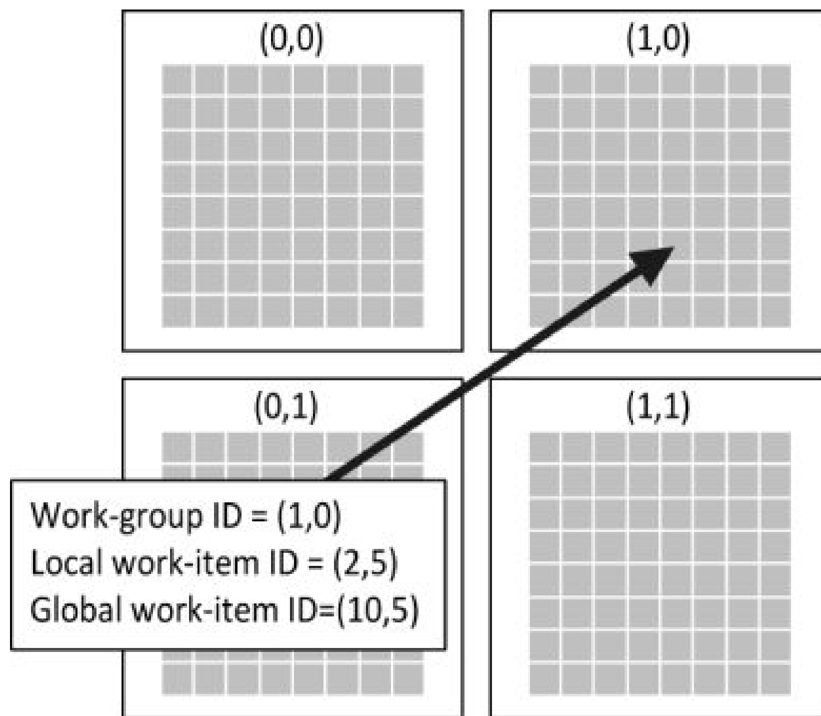
Carte de la mémoire

```
typedef struct HydroMemoryMap_s {

  // MPI domain (without overlap domains)
  long mpi_nx, mpi_ny;
  ...

  // Godunov buffer
  long gstart_q, gstart_dq, gstart_qxm, gstart_qxp;
  long gstart_c, gstart_qleft, gstart_qright;
  long gstart_qgdnv, gstart_flux;
  ...
  long godunov_total_size;

  // I x J x Var arrays, for result
  ...
} HydroMemoryMap_t;
 ...

 // Starting indexes
 hMemMapHost.gstart_q = size; size +=IJ_Var_array_size;
 hMemMapHost.gstart_c = size; size +=IJ_array_size;
 ...
 hMemMapHost.godunov_total_size = size;
 ...
```

Buffer “Godonov”

```
cl_mem godunovBufferDevice;
godunovBufferDevice = oclCAlloc( hContext, hQueue,
                          hMemMapHost.godunov_total_size,...
```

1. Introduction
   - Motivation
   - Paradigme de programmation
2. OpenCL- généralités
3. Organisation Hydro hybride
   - Modèle mémoire, modèle d'exécution
   - Gestion de la mémoire avec l'accélérateur
4. Implémentation hybride MPI/OpenMP/OpenCL
   - Modèle d'exécution (kernels)
   - Initialisation d'OpenCL
5. Conclusion
   - Performances



Ce qu'il faut savoir :

- *kernel* : source C99 compilé par l'API OpenCL sur le *host*, puis télé-chargé sur la carte accélératrice
- chaque processeur ou *Processing Unit* (PU) exécute une(des) instruction(s) SIMD sur des *Processing Unit* (PE)
- Le *Work-group* et le *Work-item* (abstrait) décrivent comment sont répartis, distribués les tableaux (1D, 2D, 3D)
- accès à des *built-in* pour savoir quel élément d'une vecteur, ... le *Work-item* traiter

Implémentation d'un *kernel* : penser que l'on ne traite qu'un point du maillage, qu'il faut localiser

Comment écrire un *kernel* ?

- Idéalement faire un *kernel* pour tout le calcul (`godunov`)
- Cependant, il y a le problème des dépendances. Par exemple, le calcul de *dq* (`slope`) ;

  ```
  dq(i) = q(i) - q(i-1)
  ```

  → les `q(i)` doivent être tous calculés avant
- Il n'y a pas de *barrier* dans un *kernel* `OpenCL`
  - *barrier* globale contraire au modèle d'exécution
  - *barrier* au niveau du *Work-group*
  - *barrier* "naturelle" ; la fin d'exécution du *kernel*
  - Autres outils de synchronisation : opérations atomiques (`atomic_xxx`)
- Il y a d'autres dépendances
  - `trace`, `riemann`, `update_U`

Éviter de trop nombreux *kernels* pour minimiser les accès à la mémoire
. . . Si dépendances, alors on coupe le *kernel* ... il y a d'autres solutions beaucoup plus performantes

Implémentation hybride MPI/OpenMP/OpenCL

Les *kernels*

Coté Fortran

```
subroutine godunov(idim,dt)
  use hydro_commons
  use OCL_interface
  ...
  if (idim==1)then
    ! Allocate work space for 1D sweeps
    call allocate_work_space(iminloc-2,imaxloc+2)

    do j=jminloc,jmaxloc
      ! Gather conservative variables
      do i=iminloc-2,imaxloc+2
        u(i,ID)=uold(i,j,ID)
        ...
      end do
      ...

      ! Convert to primitive variables
      ! inlined call constoprim(u,q,c)
      allocate(e(ijmin:ijmax))

      do i = ijmin, ijmax
        q(i,ID) = max(u(i,ID),smallr)
        q(i,IU) = u(i,IU)/q(i,ID)
        q(i,IV) = u(i,IV)/q(i,ID)
        eken = half*(q(i,IU)**2+q(i,IV)**2)
        q(i,IP) = u(i,IP)/q(i,ID) - eken
      end do

      do i = ijmin, ijmax
        e(i)=q(i,IP)
      end do

      call eos(q,e,c)
      deallocate(e)
      ! end inlined subroutine constoprim
```

Coté *kernel*

```
void __kernel kGodunovPrimVar(
          __global HydroConfig_t *hConfig,
          ...
          __global real_t *uold, int axe, real_t dt){

  real_t dx     = hConfig->dx;
  ...
  size_t i = get_global_id(0);  // Global  coordinates
  size_t j = get_global_id(1);  // of the 2D grid point

  hConfig->dtdx = dtdx;
  ...
  real_t ui[NVarMax];
  if( (i < dimX) && (j < dimY) ) {
    if (axe == 1) {
      ui[ID] = uold[ WhIJV( i, j, ID) ];
      ...
    }
    ...
    // call constoprim(u,q,c) inlined
    real_t qi[NVarMax];
    qi[ID] = fmax(ui[ID],smallr);
    qi[IU] = ui[IU]/qi[ID];
    qi[IV] = ui[IV]/qi[ID];
    real_t eken = half*(Square(qi[IU]) + Square(qi[IV]));
    qi[IP] = ui[IP]/qi[ID] - eken ;
    ...
    EquationOfStates( qi, ei, ci );

    // call constoprim(u,q,c) inlined
    __global real_t *q = &GodunovBuff[hMemMap->gstart_q];
    q[ WhIJV( i, j, ID) ] = qi[ID];
    ...
  }
}
```

- attributs explicites de placement mémoire des variables : `__global`, `__local`, `__private` (défaut)
- les boucles externes disparaissent
- correspondance `Fortran/C99`
  `Var(i,j,ID)` $\rightarrow$ `Varij[ID]` (on perd une dimension ou plus)
- les arguments du *kernel* ne sont pas copiés
- version `CUDA` meilleur intégration mais proche

Implémentation hybride MPI/OpenMP/OpenCL

Lancement des *kernels*

exécution d'un *kernel*

```
void ocl_godunov_( int *iaxe, real_t *dt, int *nstep ) {

  size_t globalWorkSize[] = {dimX, dimY, 0};

  int count=0;
  cl_kernel k = hKernels[godunovPrimVar];
  clSetKernelArg( k, count++, ..., &hydroConfigDevice);
  clSetKernelArg( k, count++, ..., &hydroMemMapDevice);
  clSetKernelArg( k, count++, ..., &godunovBufferDevice);
  clSetKernelArg( k, count++, ..., &uoldDevice);
  clSetKernelArg( k, count++, ..., iaxe);
  clSetKernelArg( k, count++, ..., dt);

  oclLaunchKernelAndWait( hQueue, k,
                          WorkDim2D, ZeroOffset,
                          globalWorkSize, AutoLocalWorkSize
                        );
  ...

  oclLaunchKernelAndWait( hQueue, k,
                          WorkDim2D, ZeroOffset,
                          globalWorkSize, AutoLocalWorkSize
                        );
  ...
}
```

- `ocl_godunov_` appelé depuis `module_hydro_principal.f90`
- `globalWorkSize` grille, domaine à distribuer
- `hKernels[godunovPrimVar]` sélection du *kernel*
- `clSetKernelArg` empilement des paramètres du *kernel* (doivent correspondre)
- `oclLaunchKernelAndWait` lancement du *kernel*

```
clSetKernelArg( k, count++, sizeof(cl_mem), (void *) &hydroConfigDevice);
```

Variables importantes

```
// Host buffers
real_t *uoldHost;
...
// Device buffers
cl_mem  uoldDevice;
...
// OCL handlers
...
// Kernel IDs
....
// OCL handlers, specific to Hydro code
cl_context        hContext;
cl_command_queue  hQueue;
cl_kernel        *hKernels;
...
// Kernel default worksize
size_t *localWorkSize = NULL;
```

- Correspondance des variables entre les 2 espaces mémoires `Host` et `Device`
- Une fois l'initialisation `OpenCL` faite, plus qu'à soumettre des *kernels*, des ordres de copie à la queue `hQueue`
- tableaux de *kernels* `hKernels[godunovPrimVar]`
- `localWorkSize`, on laisse le *device* choisir la meilleur taille

Appel depuis Fortran

```
void ocl_init_hydro_( int *mpi_rank, ...)
  // Init OpenCL
  oclInit(*mpi_rank);

  // Select the OCL queues
  hContext = contexts[0].context;
  hQueue   = contexts[0].queues[0];
  hKernels = contexts[0].kernels;

  // Configure the memory map (distribution parameters,
          ...
  hydroMemMap.mpi_nx    = *mpi_nx;
  ...
  // Config
  hydroConfig.nstepmax = *nstepmax;
  ...
  // Alloc Hydro configuration on the device
  hydroConfigDevice = oclAlloc( hContext, sizeof(
         HydroConfig_t));
  ...      // idem buffer MemMap, Uold, Results
  // Build memory map for Godunov arrays and allocate
         the buffers
  allocateGodunovWorkSpace();
  ...

  // Copy to device
  oclCopyHostToDevice( hQueue, &hydroConfig,
         hydroConfigDevice, sizeof(HydroConfig_t) );
  ...     // hydroMemMap ->hydroMemMapDevice

  ocl_init_uold();
}
```

- fonctions `clxxxx` → standard `OpenCL`
- fonctions `oclxxxx` → API utilitaire (`LLR/GridCL`)
- initialisation de `OpenCL`
- sélection de la queue parmi celles disponibles (à l'exécution de `hydro` les *devices* trouvés sont décrits dans le fichier `config.acc`)
- créations des *buffers* dans le(s) *device(s)*, puis copies *Host* → *Device*
- initialisation de `Uold` dans le *device* (*kernel*)

Implémentation hybride MPI/OpenMP/OpenCL

Initialisation

OpenCL initialisation

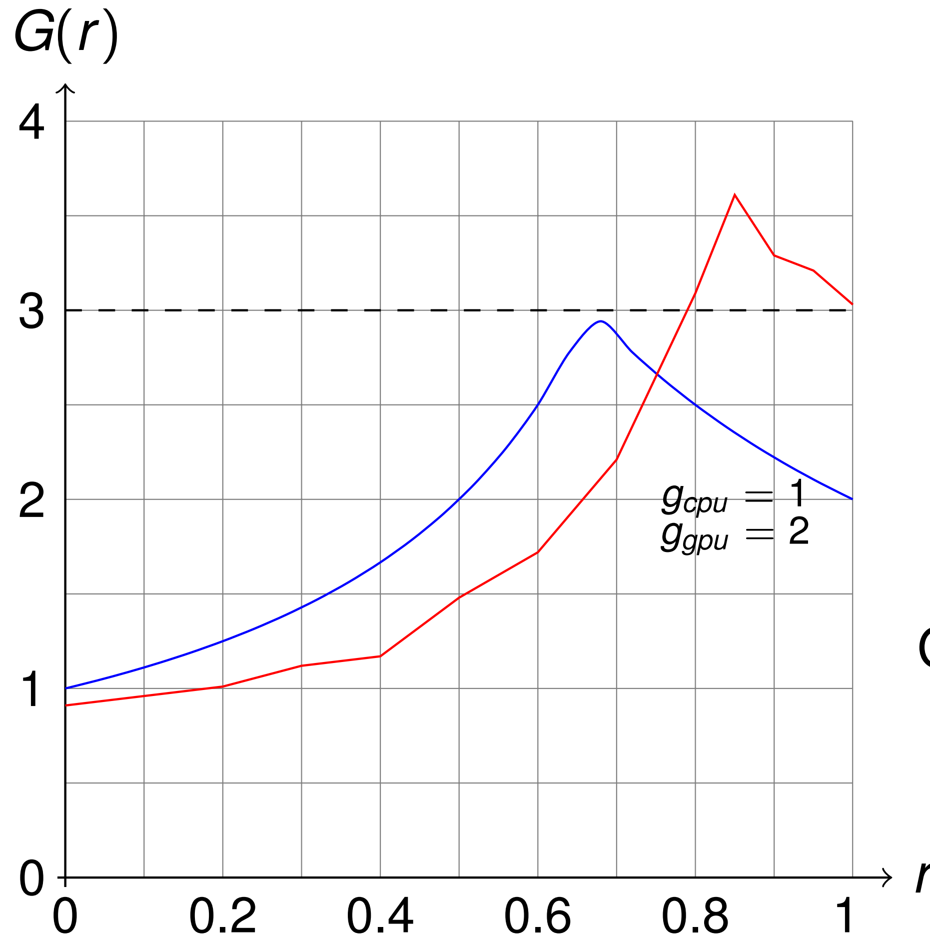
```
void oclInit( int mpi_rank) {
 ...
 // Scan the available platforms
 platformSet = oclDiscoverPlatforms( CL_DEVICE_TYPE_GPU,
        configFile );

 // Select the platform
 ...
 selectedPlatforms[0] = 0;

 // Make the plaform contexts (one queue per device)
 contexts = oclMakeContextsAndQueues( platformSet, ...)

 // Build kernels and kernel libraries
 ...
 oclMakeProgramsAndBuildKernels( contexts, ... )
 ...
}

void allocateGodunovWorkSpace() {
 // Starting indexes
 hydroMemMap.gstart_q = size; size += hydroMemMap.
        I_J_Var_array_size;
 ...
 hydroMemMap.godunov_total_size = size;

 godunovBufferDevice = oclCAlloc( hContext, hQueue, ... )
        ;
 ...
}

// Fortran calls - kernels  execution
// acc -> host
void get_device_uold_boundaries_v2_( int *axe_, real_t *
       f90Uold) {
...
```

- balayage des plateformes disponibles (`config.acc`)
- autres filtres : `CL_DEVICE_TYPE_CPU, ..._ALL`
- sélection d'une ou plusieurs plateformes
- création des *contexts* et des *queues*
- création, compilation des `kernels`
- allocations dans le *device* du *buffer* “Godunov” via la “carte mémoire”, puis copie
- mise à jour des frontières de `Uold` : `get_device_uold_boundaries_v2_`

1. Introduction
   - Motivation
   - Paradigme de programmation
2. `OpenCL`- généralités
3. Organisation `Hydro` hybride
   - Modèle mémoire, modèle d'exécution
   - Gestion de la mémoire avec l'accélérateur
4. Implémentation hybride `MPI/OpenMP/OpenCL`
   - Modèle d'exécution (kernels)
   - Initialisation d'`OpenCL`
5. Conclusion
   - Performances

Implémentation hybride MPI/OpenMP/OpenCL

Performances



- configuration 5000×5000
- CC-IN2P3 : Intel E5-2670 + NVidia M2090
- fraction GPGPU dans module_omp_commun.f90 : thread_domain_ratio = 0.5
- temps minimun sur 2 exécution pour un ratio.

Gain :

$$G = \frac{1}{max\{(1-r)/g_{cpu},\ r/g_{gpu}\}}$$

$g_{cpu,gpu}$ performances des CPU et GPGPU

r fraction du domaine GPGPU

Conclusion

Perspectives

Bilan :

- aperçu rapide de la version hybride OpenCL
- mais, on a vu tous les points majeurs
- de même, les principes d'OpenCL

Optimisations restantes :

- Réduction pour calculer *dt*
- Faire un seul *kernel* de calcul

Parties manquantes :

- déploiement sur Xeon Phi sans modification
- démonstration que OpenCL optimise sur CPU les codes CPU