



# **Rencontres Arcane 2025 : API Accélérateur**

24 mars 2025

Gilles Grospellier, CEA, DAM, DIF, F-91297 Arpajon Cedex

# Plan

Contexte

Choix de conception

Description des fonctionnalités

Performances

Conclusion

# Contexte

- Arcane est l'architecture de plusieurs codes de calcul du CEA/DAM
  - Environ 3 millions de lignes de C++ pour ces codes
  - Plusieurs physiques, multi-matériaux, différents types de maillages (cartésien, AMR, non structuré)
  - Plus de 10000 boucles de calcul par code : pas de point chaud au niveau du profilage
- Besoin de porter nos codes sur accélérateurs
  - **Évolution incrémentale indispensable**
  - Perturber le moins possible le code source
    - Même code source pour le CPU et GPU si possible
    - L'aspect validation est primordial
  - Solution souveraine et pérenne

# Conception de l'API Accélérateur

- Un seul code et **un seul exécutable** pour le CPU et le GPU
  - Pouvoir choisir dynamiquement (sans recompilation) de tourner sur CPU ou GPU (ou un mélange des deux)
  - Possibilité d'avoir des sous-domaines sur GPU et d'autres sur CPU
  - Possibilité d'utiliser du dataflow pour gérer les copies entre CPU et GPU
- API spécialisée pour les concepts Arcane permettant d'utiliser sur GPU les éléments suivants
  - Variables
  - Boucles sur les entités du maillage
  - Connectivités (Nœuds, mailles, ...) et constituants (Matériaux, Milieux) du maillage
- Abstraction des fonctionnalités pour cacher l'implémentation sous-jacente au code utilisateur
- Pas d'utilisation de mécanisme dépendant du compilateur (OpenMP target, OpenACC, ...) pour améliorer la portabilité

# Implémentation

- Début des développements fin 2020
- API similaire à SYCL car c'est l'API la plus portable
- Utilisation via les fonctions lambda du C++11
- Utilisation par défaut de la mémoire managée pour simplifier les transferts mémoire entre CPU et GPU
  - Il est possible de gérer soit même la mémoire (Device) si on le souhaite
- Accès possible aux structures spécifiques à la plateforme (par exemple `cudaStream_t`) en sacrifiant la portabilité : Couplage possible avec d'autres API (CUDA, ROCM, RAJA, Kokkos, YAKL, ...)

# Implémentation

- Limite au maximum l'utilisation de `template` du C++
  - Code plus simple pour les non spécialistes du C++
  - Facilite le développement
    - Compilation beaucoup plus rapide
    - Évite de recompiler une grosse partie du code à chaque changement
    - Permet de changer dynamiquement le comportement (par exemple quelle mémoire utiliser)
  - Code plus facilement inter-opérable avec d'autres bibliothèques (MPI, IO, ...)

# Exemple : Boucle pour initialiser un gaz parfait

## ■ Déclaration des variables

```
VariableCellReal m_density; //!< Density on cells
VariableCellReal m_pressure; //!< Pressure on cells
VariableCellReal m_sound_speed; //!< Sound speed on cells
VariableCellReal m_internal_energy; //!< Internal energy on cells
VariableCellReal m_adiabatic_cst; //!< Adiabatic constant on cells
```

## ■ Itération sur toutes les mailles

- `vi` est l'itérateur, `ENUMERATE_CELL` la macro pour itérer, `allCells()` la liste des mailles

```
// Initialise l'énergie et la vitesse du son
ENUMERATE_CELL(vi, allCells())
{
    Real pressure = m_pressure[vi];
    Real adiabatic_cst = m_adiabatic_cst[vi];
    Real density = m_density[vi];
    m_internal_energy[vi] = pressure / ((adiabatic_cst - 1.) * density);
    m_sound_speed[vi] = sqrt(adiabatic_cst * pressure / density);
}
```

# Exemple : Boucle pour initialiser un gaz parfait

## Écriture classique

```
// Initialise l'énergie et la vitesse du son
ENUMERATE_CELL(vi, allCells())
{
    Real pressure = m_pressure[vi];
    Real adiabatic_cst = m_adiabatic_cst[vi];
    Real density = m_density[vi];
    m_internal_energy[vi] = pressure / ((adiabatic_cst - 1.) * density);
    m_sound_speed[vi] = sqrt(adiabatic_cst * pressure / density);
}
```

- Même structure de boucle avec des modifications mineures
  - Utilise RUNCOMMAND\_ENUMERATE au lieu de ENUMERATE
  - Le code est une lambda du C++11 (voir le ‘;’ à la fin de la boucle)

## Code avec API accélérateur

```
info() << "Initialize SoundSpeed and InternalEnergy";
auto queue = makeQueue(m_runner);
auto command = makeCommand(queue);
// Initialise l'énergie et la vitesse du son
auto in_pressure = ax::viewIn(command, m_pressure);
auto in_density = ax::viewIn(command, m_density);
auto in_adiabatic_cst = ax::viewIn(command, m_adiabatic_cst);

auto out_internal_energy = ax::viewOut(command, m_internal_energy);
auto out_sound_speed = ax::viewOut(command, m_sound_speed);
```

```
command << RUNCOMMAND_ENUMERATE(Cell, vi, allCells())
{
    Real pressure = in_pressure[vi];
    Real adiabatic_cst = in_adiabatic_cst[vi];
    Real density = in_density[vi];
    out_internal_energy[vi] = pressure / ((adiabatic_cst-1.0) * density);
    out_sound_speed[vi] = math::sqrt(adiabatic_cst*pressure/density);
};
```



# Exemple : Boucle pour initialiser un gaz parfait

## Écriture classique

```
// Initialise l'énergie et la vitesse du son
ENUMERATE_CELL(vi, allCells())
{
    Real pressure = m_pressure[vi];
    Real adiabatic_cst = m_adiabatic_cst[vi];
    Real density = m_density[vi];
    m_internal_energy[vi] = pressure / ((adiabatic_cst - 1.) * density);
    m_sound_speed[vi] = sqrt(adiabatic_cst * pressure / density);
}
```

- Il faut indiquer l'intention
  - `viewIn()` si lecture seule
  - `viewOut()` si écriture seule
  - `viewInOut()` si lecture/écriture
- Copie automatiquement des données entre le GPU et le CPU
  - Pas forcément nécessaire si on utilise la mémoire unifiée

## Code avec API accélérateur

```
info() << "Initialize SoundSpeed and InternalEnergy";
auto queue = makeQueue(m_runner);
auto command = makeCommand(queue);
// Initialise l'énergie et la vitesse du son
```

```
auto in_pressure = ax::viewIn(command, m_pressure);
auto in_density = ax::viewIn(command, m_density);
auto in_adiabatic_cst = ax::viewIn(command, m_adiabatic_cst);
```

**ENTRÉES**

```
auto out_internal_energy = ax::viewOut(command, m_internal_energy);
auto out_sound_speed = ax::viewOut(command, m_sound_speed);
```

**SORTIES**

```
command << RUNCOMMAND_ENUMERATE(Cell, vi, allCells())
{
    Real pressure = in_pressure[vi];
    Real adiabatic_cst = in_adiabatic_cst[vi];
    Real density = in_density[vi];
    out_internal_energy[vi] = pressure / ((adiabatic_cst-1.0) * density);
    out_sound_speed[vi] = math::sqrt(adiabatic_cst*pressure/density);
};
```

# Exemple : Boucle pour initialiser un gaz parfait

## Écriture classique

```
// Initialise l'énergie et la vitesse du son
ENUMERATE_CELL(vi, allCells())
{
    Real pressure = m_pressure[vi];
    Real adiabatic_cst = m_adiabatic_cst[vi];
    Real density = m_density[vi];
    m_internal_energy[vi] = pressure / ((adiabatic_cst - 1.) * density);
    m_sound_speed[vi] = sqrt(adiabatic_cst * pressure / density);
}
```

- Utilise les abstractions suivantes
  - `m_runner` est la ressource d'exécution (CPU, GPU, multi-thread, ...)
  - `queue` est le flot d'exécution
  - `command` représente un noyau de calcul

## Code avec API accélérateur

```
info() << "Initialize SoundSpeed and InternalEnergy";
auto queue = makeQueue(m_runner);
auto command = makeCommand(queue);
// Initialise l'énergie et la vitesse du son
auto in_pressure = ax::viewIn(command, m_pressure);
auto in_density = ax::viewIn(command, m_density);
auto in_adiabatic_cst = ax::viewIn(command, m_adiabatic_cst);

auto out_internal_energy = ax::viewOut(command, m_internal_energy);
auto out_sound_speed = ax::viewOut(command, m_sound_speed);

command << RUNCOMMAND_ENUMERATE(Cell, vi, allCells())
{
    Real pressure = in_pressure[vi];
    Real adiabatic_cst = in_adiabatic_cst[vi];
    Real density = in_density[vi];
    out_internal_energy[vi] = pressure / ((adiabatic_cst-1.0) * density);
    out_sound_speed[vi] = math::sqrt(adiabatic_cst*pressure/density);
};
```

# État des développements

- Implémentation disponible pour fonctionnalités suivantes
  - Gestion de la connectivité non structurée et cartésienne
  - Gestion multi-matériau
  - Gestion de l'asynchronisme
- Algorithmes haut niveau : Réductions, PrefixSum, Filtrage, Partitionnement de liste, Tri
- Classe `NumArray`, pour gérer les tableaux jusqu'à 4 dimensions
- Support des implémentations 'MPI Accelerator Aware'
  - Échange de message
  - Synchronisations
- Mécanismes d'aide au développement

# Aide au développement

- Profilage automatique des boucles
  - Il suffit de positionner la variable d'environnement `ARCANE_LOOP_PROFILING_LEVEL`
  - Fonctionne pour toutes les boucles Arcane (`RUNCOMMAND_...`, `ENUMERATE_...`)
  - Fonctionne pour tous les modes: séquentiel, multi-thread, accélérateur.
- Intégration avec CUPTI (Cuda Profiling Tools interface)
  - Bibliothèque NVIDIA permettant de récupérer les évènements sur les GPU NVIDIA
  - Dans Arcane, permet de récupérer les évènements suivants
    - Copies mémoire managée entre le GPU et le CPU
    - Lancement des noyaux de calcul
  - Permet de tracer les noyaux qui effectuent des transferts mémoire
  - Activation automatique par variable d'environnement

# Démonstrateurs

- Dans Arcane
  - MicroHydro (1000 lignes de code)
  - MiniWeather (<https://github.com/mrnorman/miniWeather.git>) (800 lignes de code)
- MaHyCo (<https://github.com/cea-hpc/MaHyCo>) (10000 lignes de code)
  - Hydrodynamique multi-fluide
- Athena (20k lignes de code)
  - Code interne CEA/DAM pour évaluer les schémas de transport et de diffusion
  - Équation de transport prototype de neutronique, résolue par une méthode déterministe et développement en cours sur des méthodes Monte-Carlo
  - Utilisé activement par les stagiaires pour mener des études (3 stagiaires en 2023)
- Support expérimental des accélérateurs de l'API éléments finis d'Arcane
  - <https://github.com/arcaneframework/arcanefem>

# Profilage

- Profilage automatique en positionnant la variable d'environnement ARCANES\_LOOP\_PROFILING\_LEVEL

- MicroHydro avec MPI non cuda-aware

Synchronisation

Ncall	Nchunk	T (ms)	Tck (ns)	%	name
100	0	1069.990	0	55.9	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeVelocity()
101	0	403.066	0	21.0	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeGeometricValues()
200	0	272.795	0	14.2	void SimpleHydro::SimpleHydroAcceleratorService::_computePressureAndCellPseudoViscosityForces()
500	0	47.554	0	2.4	virtual void SimpleHydro::SimpleHydroAcceleratorService::applyBoundaryCondition()
100	0	38.912	0	2.0	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeViscosityWork()
1	0	25.234	0	1.3	virtual void SimpleHydro::SimpleHydroAcceleratorService::hydroStartInit()
100	0	16.325	0	0.8	virtual void SimpleHydro::SimpleHydroAcceleratorService::updateDensity()
100	0	16.317	0	0.8	virtual void SimpleHydro::SimpleHydroAcceleratorService::applyEquationOfState()
100	0	14.876	0	0.7	virtual void SimpleHydro::SimpleHydroAcceleratorService::moveNodes()
100	0	6.196	0	0.3	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeDeltaT()

- MicroHydro avec MPI cuda-aware

Synchronisation  
(reception)

Ncall	Nchunk	T (ms)	Tck (ns)	%	name
101	0	370.668	0	44.7	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeGeometricValues()
200	0	272.100	0	32.8	void SimpleHydro::SimpleHydroAcceleratorService::_computePressureAndCellPseudoViscosityForces()
100	0	37.859	0	4.5	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeViscosityWork()
1	0	24.210	0	2.9	virtual void SimpleHydro::SimpleHydroAcceleratorService::hydroStartInit()
101	0	20.148	0	2.4	void Arcane::Accelerator::impl::AcceleratorSpecificMemoryCopy<DataType, Extent>::_copyFrom()
3	0	17.222	0	2.0	void Arcane::Accelerator::impl::AcceleratorSpecificMemoryCopy<DataType, Extent>::_copyFrom()
100	0	16.359	0	1.9	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeVelocity()
100	0	16.212	0	1.9	virtual void SimpleHydro::SimpleHydroAcceleratorService::applyEquationOfState()
100	0	15.752	0	1.9	virtual void SimpleHydro::SimpleHydroAcceleratorService::updateDensity()
100	0	14.761	0	1.7	virtual void SimpleHydro::SimpleHydroAcceleratorService::moveNodes()
500	0	12.343	0	1.4	virtual void SimpleHydro::SimpleHydroAcceleratorService::applyBoundaryCondition()
100	0	6.333	0	0.7	virtual void SimpleHydro::SimpleHydroAcceleratorService::computeDeltaT()

Synchronisation  
(envoi)

3	0	1.993	0	0.2	void Arcane::Accelerator::impl::AcceleratorSpecificMemoryCopy<DataType, Extent>::_copyTo()
101	0	1.562	0	0.1	void Arcane::Accelerator::impl::AcceleratorSpecificMemoryCopy<DataType, Extent>::_copyTo()

# Conclusion

- L'API accélérateur est opérationnelle pour plusieurs applications internes
  - Implémentation disponible à 100% pour CUDA et ROCM
  - Utilisation de oneTBB pour le multi-threading
  - 12000 lignes de code au total
  - <https://github.com/arcaneframework>
- Prochains développements
  - Portage SYCL/DPC++ (90% réalisé mais non prioritaire)
  - Support du parallélisme hiérarchique
  - Gestion des tableaux associatifs (`std::unordered_map`)
  - Ajouts de classes spécifiques pour gérer des filtres sur les mailles ou les constituants



**Merci de  
votre  
attention**

