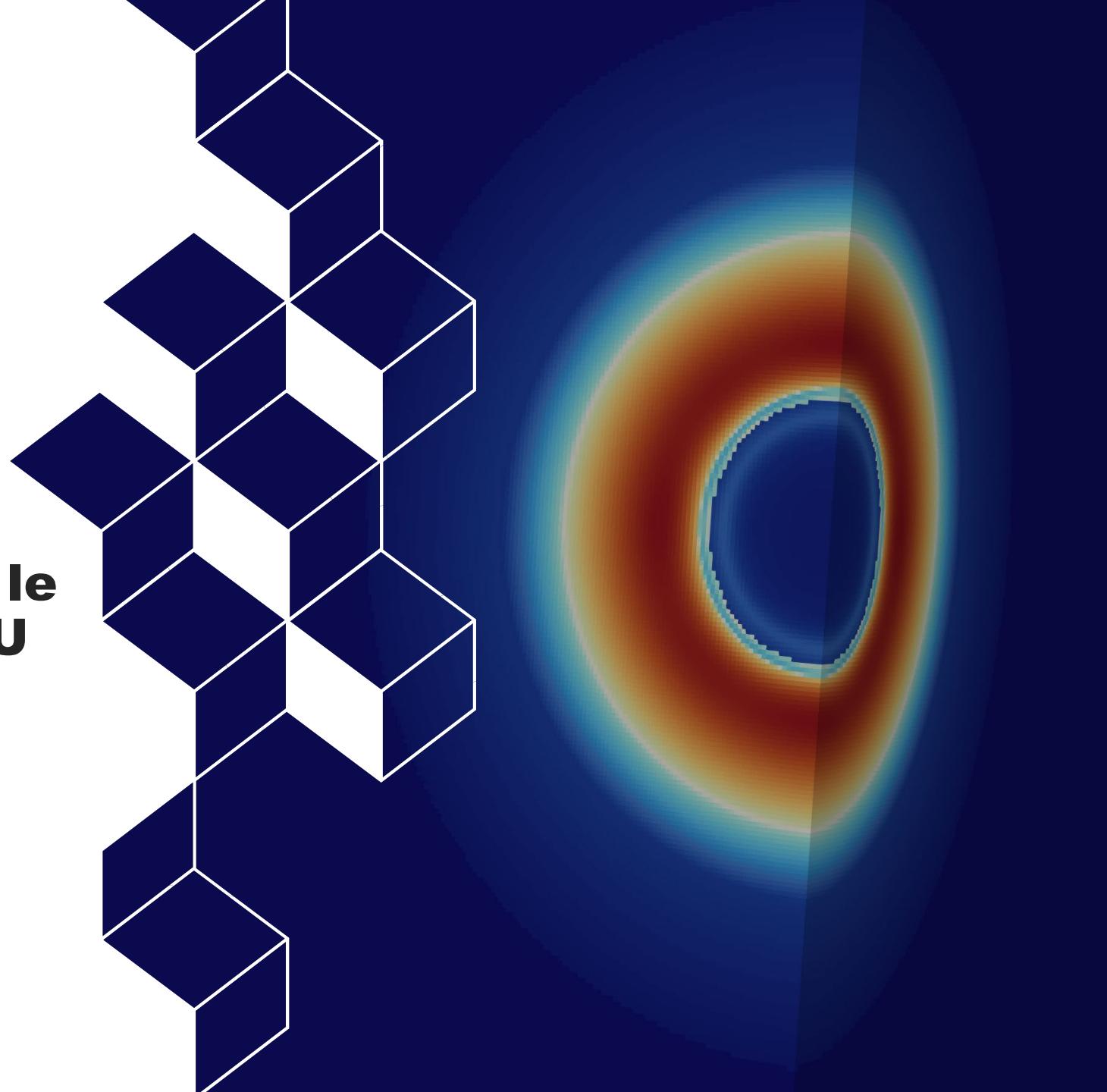




Utilisation de l'API accélérateur Arcane pour le portage de Phénix sur GPU

Simon Pomarède



Sommaire

- 1. Présentation de Phénix et du contexte de portage GPU**
- 2. Outils mis en œuvre pour le portage GPU**
- 3. Exemple de portage : méthode de Newton pour les équations d'état**
- 4. Performances**
- 5. Pour conclure**



1

Présentation de Phénix et du contexte de portage GPU



Présentation de Phénix

Code multi-physique de dynamique rapide 2D-3D massivement parallèle :

- Des physiques intimement couplées
 - Hydrodynamique compressible (équations d'Euler), élasto-plasticité, détonique, rupture et endommagement, ...
 - Couplage aux bibliothèques de lois physiques (EOS, rhéologie)
- Des méthodes numériques adaptées
 - Volumes finis, Contexte multi-matériaux
 - Lagrangienne, ALE sur maillage non structuré : schéma VNR + projection multidimensionnel
 - Eulérienne sur maillage structuré : schéma Lagrange (prédicteur-correcteur) + projection par directions alternées
- Parallélisme par décomposition de domaine (MPI)

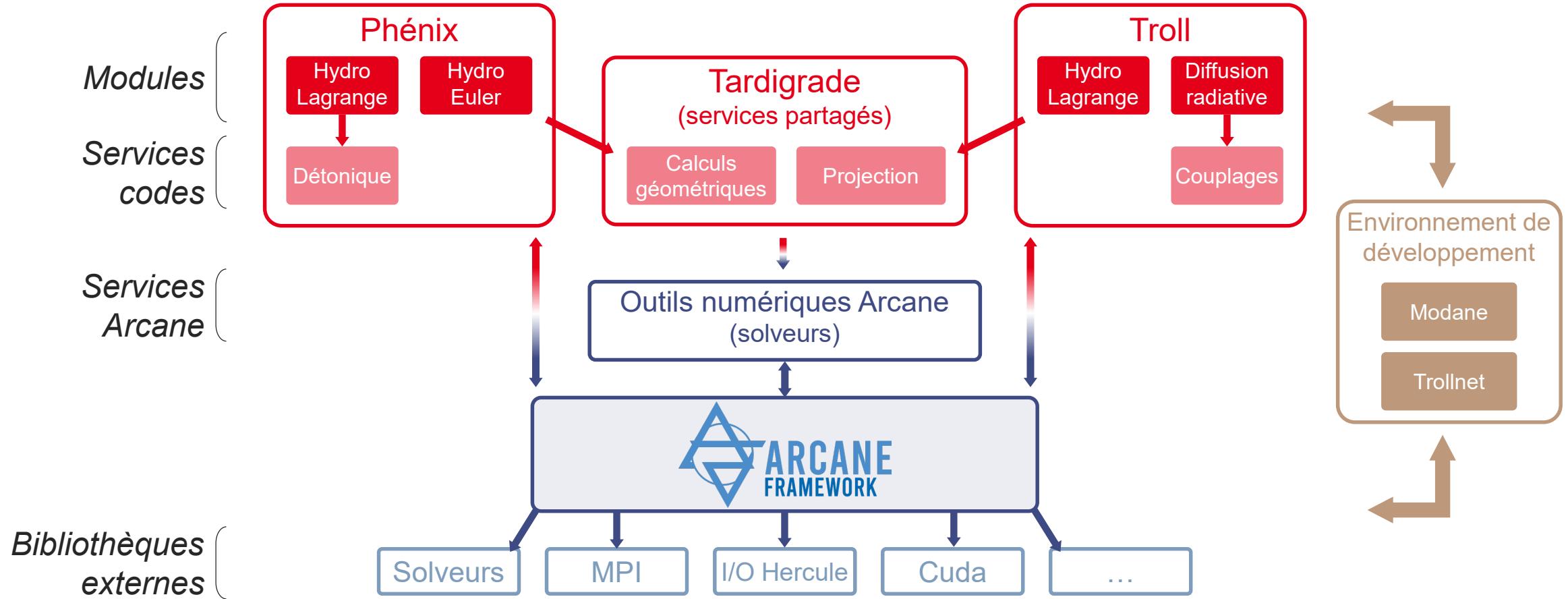
Développé depuis 2012

- Codé en C++ sur la plateforme [Arcane](#), 1 700 000 lignes
- Equipe de développement d'une quinzaine d'ingénieurs, répartis entre la DIF et le CESTA (Aquitaine)
- Remplacement de deux codes historiques (fin années 90)

En production depuis 2018

- ~100 utilisateurs sur trois centres, pour 460 millions d'heures de calculs en 2024

Fonctionnalités Arcane utilisées par Phénix





Contexte de portage GPU

Equipe de développement

- Laboratoire de développement de Phénix : équipe dédiée au portage GPU
- Équipe de spécialistes (DSSI, comme équipe [Arcane](#))
- Équipe Arcane (évolutions d'[Arcane](#) en fonction de nos besoins)

Historique du portage

- Portage commencé en 2021
- Développements dans MaHyCo, code ouvert basé sur [Arcane](#)
 - Mise au point de l'API Arcane
 - Monté en compétence de l'équipe de développement
 - Permet des discussions avec NVidia
- Depuis 2022 : portage effectif du code
- Besoin d'une version du code qui puisse s'exécuter sur les machines GPU de la DAM

Cohabitation des services CPU & Acc

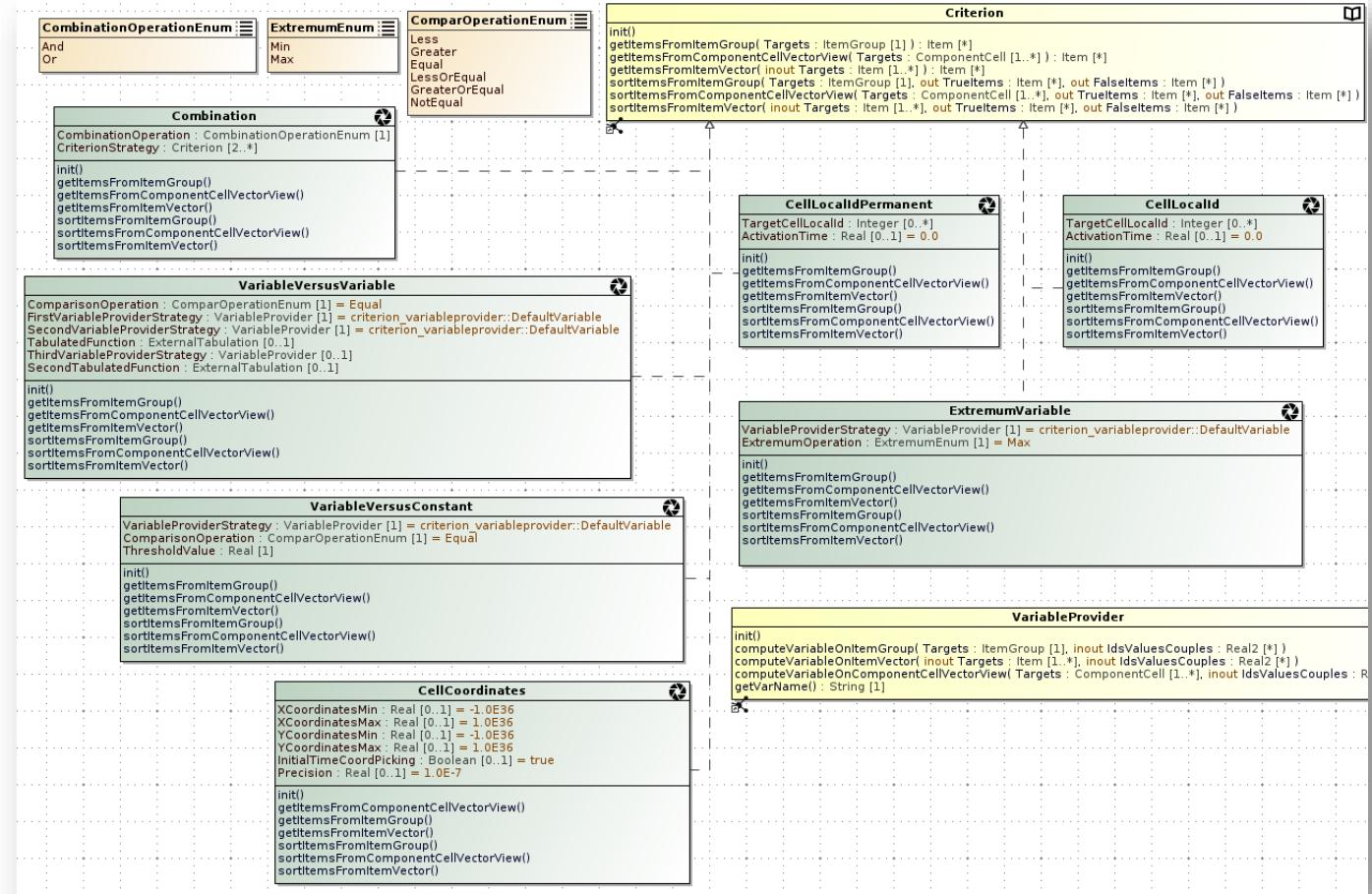
- Génération par Modane des services classiques et Acc (version « accélérateurs » : GPU mais également multithread)



2 ■ Outils mis en œuvre pour le portage GPU

Modane : modélisation UML des composants

- Auparavant : modélisation avec *MagicDraw* et génération des sources par *Modane* (cf. rencontres Arcane 2023)



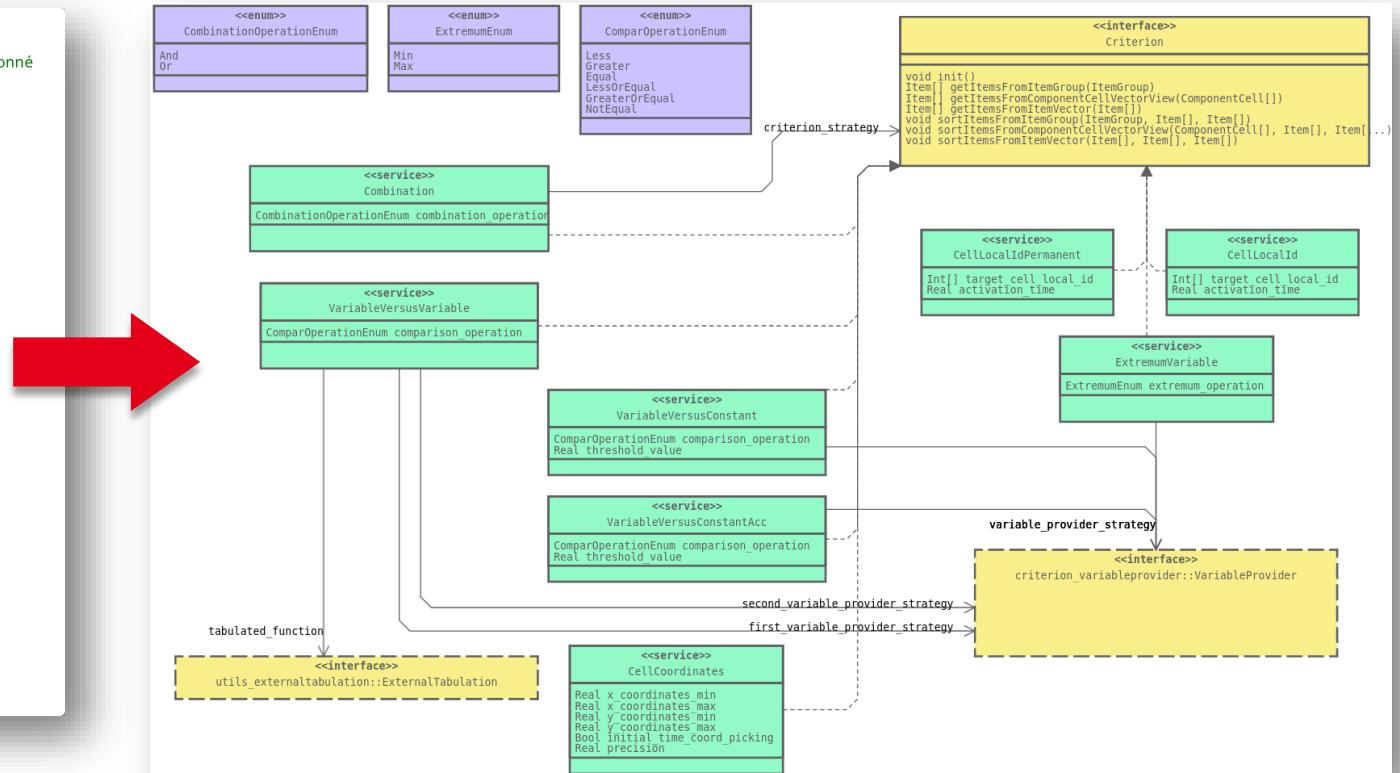
Modane : modélisation UML des composants

- Actuellement (Modane v2) :
- Version motivée par les difficultés de fusion de modèles avec MagicDraw
- Modélisation dans un format texte et représentation graphique par Modane (plugin VSCode)

```

1  /*
2   * Service critère pour allumer une seule fois un groupe de mailles en même temps, pour un temps donné
3   */
4  service caseoption CellLocalId implements Criterion
5  {
6    opt Int[*] target_cell_local_id;
7
8    opt Real activation_time = "0.0"
9    namefr "temps_activation"
10   categories CategoriesModel::User;
11
12  override void init();
13
14  override Item[*] getItemsFromItemGroup(in ItemGroup targets);
15
16  override Item[*] getItemsFromComponentCellVectorView(in ComponentCell[*] targets);
17
18  override Item[*] getItemsFromItemVector(inout Item[*] targets);
19
20  override void sortItemsFromItemGroup(in ItemGroup targets,
21    out Item[*] true_items,
22    out Item[*] false_items);
23
24
25  override void sortItemsFromComponentCellVectorView(in ComponentCell[*] targets,
26    out Item[*] true_items,
27    out Item[*] false_items);
28
29  override void sortItemsFromItemVector(inout Item[*] targets,
30    out Item[*] true_items,
31    out Item[*] false_items);
32 }

```



Modane : génération des sources

Génération d'instrumentation du code d'origine

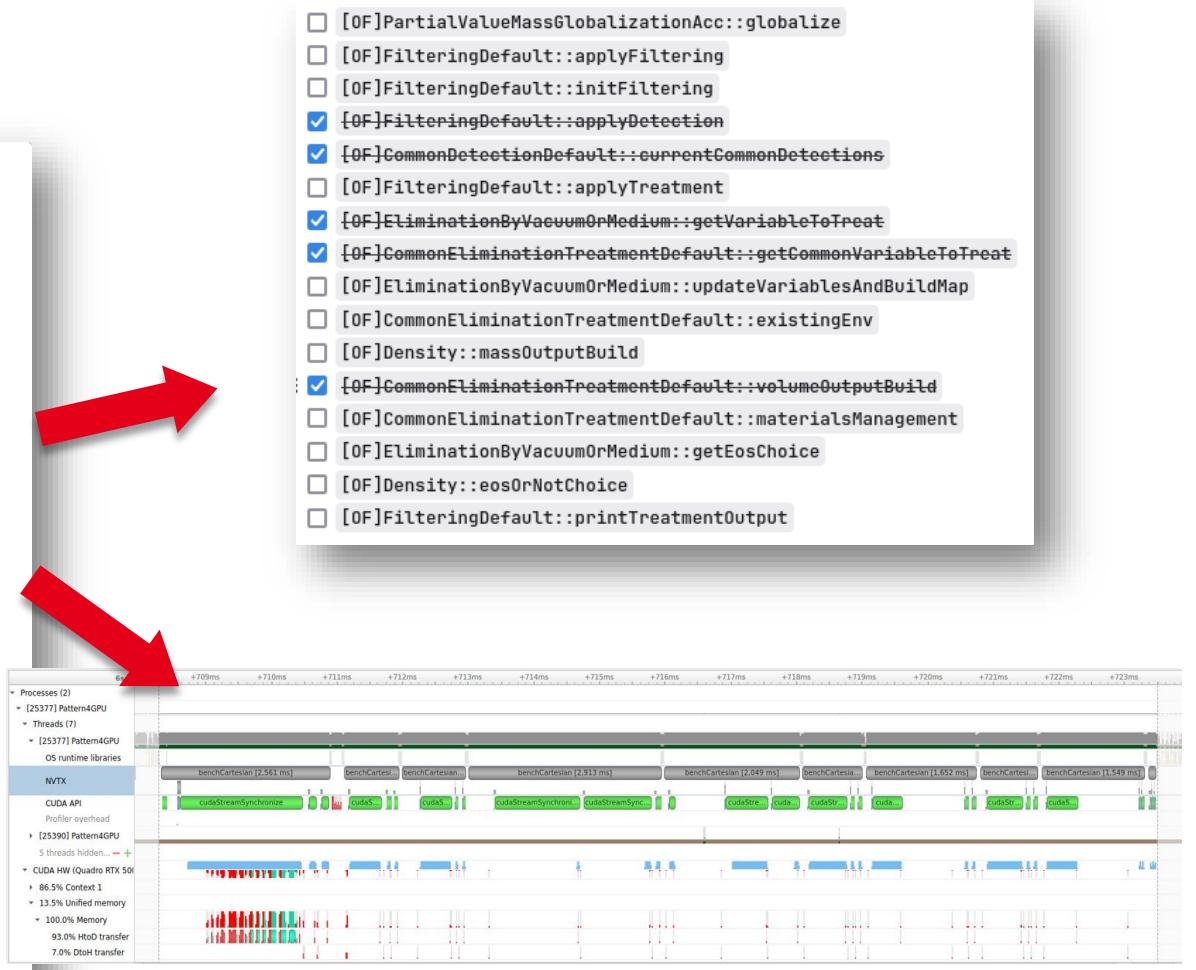
- Permet d'identifier les méthodes appelées

```

1  /*!
2   Lecture dans le jdd des coefficients de pseudo par milieu, et stockage des coefficients par environnement dans
3   m_pseudo_viscosity_coefficient
4   Cette méthode construit les variables et appelle EulHydroLagModule::beginPseudoViscosity.
5 */
6 void beginPseudoViscosity()
7 {
8     #if not defined(PROF_ACC_DISABLED)
9     prof_acc_begin("[EP]EulHydroLag::beginPseudoViscosity");
10    #endif
11
12    EulHydroLagBeginPseudoViscosityVars vars(m_pseudo_viscosity_coefficient);
13    this->beginPseudoViscosity(vars);
14    #if not defined(PROF_ACC_DISABLED)
15    prof_acc_end("[EP]EulHydroLag::beginPseudoViscosity");
16    #endif
17
18 /**
19  Initialisation du tenseur de déformation (uniquement si on veut visualiser cette grandeur) : essentiel, car il
20  est calculé après par une loi d'évolution.
21  Cette méthode construit les variables et appelle EulHydroLagModule::beginStrainTensor.
22 */
23 void beginStrainTensor()
24 {
25     #if not defined(PROF_ACC_DISABLED)
26     prof_acc_begin("[EP]EulHydroLag::beginStrainTensor");
27     #endif
28
29     EulHydroLagBeginStrainTensorVars vars;
30     this->beginStrainTensor(vars);
31     #if not defined(PROF_ACC_DISABLED)
32     prof_acc_end("[EP]EulHydroLag::beginStrainTensor");
33     #endif
}

```

Temps d'exécution



Modane : génération des sources

Maintien en cohérence de deux versions des services portés

- Services « CPU » non modifiés
- Services « Acc » créés

Génération de code spécifique accélérateurs

- Génération de code plus performant
- Génération de structures spécifiques Acc :
 - **generateForAccelerator** : génère des boucles accélérateurs pour les méthodes avec support
 - **generateLoopWrapper** : génère des *wrappers* pour effectuer des traitements sur CPU

```
{
  "generateForAccelerator": [
    "elasticity::ElasticityAcc::copyOldDeviatoricStressTensor",
    "elasticity::ElasticityAcc::computeElastoPlasticEnergy",
    "elasticity::ElasticityAcc::computeDeviatoricStressTensor",
    "elasticity::ElasticityAcc::updateSoundSpeed"
  ],
  "generateLoopWrappers": [
    "elasticity::ElasticityAcc::computeElastoPlasticEnergy",
    "elasticity::ElasticityAcc::updateSoundSpeed"
  ]
}
```



```
void updateSoundSpeed(const EnvCellVectorView items,
                      const MaterialVariableCellReal& abstract_shear_modulus,
                      const MaterialVariableCellReal& abstract_density,
                      MaterialVariableCellReal& abstract_sound_speed,
                      const Real coefficient) override
{
  auto queue = getAccEnv()->newQueue();
  auto command = makeCommand(queue);
  T* t = static_cast<T*>(this);
  ElasticityAccUpdateSoundSpeedViews<T> acc_context(t,
    ax::viewIn(command, abstract_shear_modulus),
    ax::viewIn(command, abstract_density),
    ax::viewInOut(command, abstract_sound_speed),
    coefficient);
  updateSoundSpeedBeforeLoop(abstract_shear_modulus, abstract_density, abstract_sound_speed, coefficient);
  command << RUNCOMMAND_MAT_ENUMERATE(EnvAndGlobalCell, evgi, items) {
    auto [mvi, cid] = evgi();
    acc_context.apply(mvi, cid);
  };
  updateSoundSpeedAfterLoop(abstract_shear_modulus, abstract_density, abstract_sound_speed, coefficient);
}
```



Validation du portage

Pour valider le portage en cours

Comparaison des résultats des exécutions « CPU » vs « Acc » :

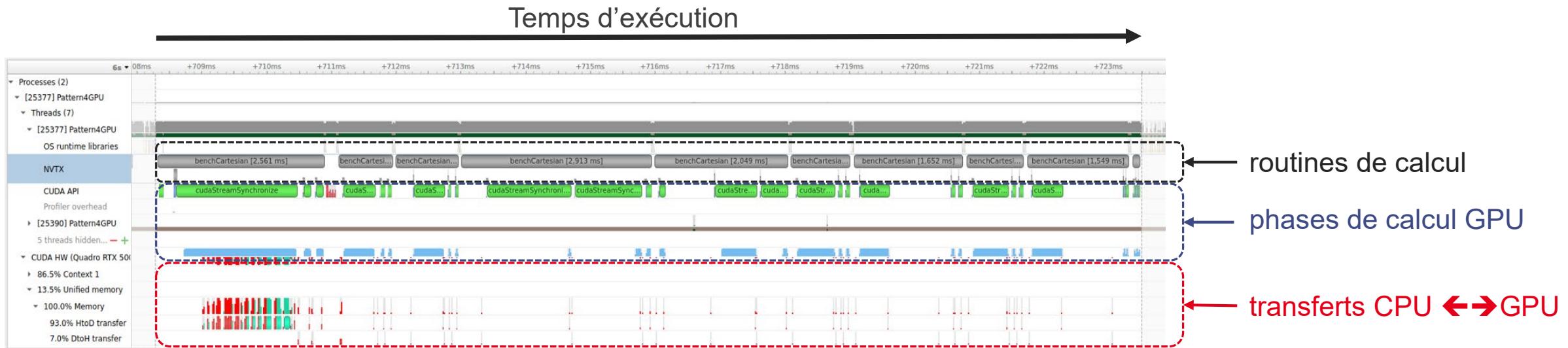
- **hverify** : outil commun (*Tardigrade*) de comparaison des sommes de contrôles des variables
 - + comparaison bit à bit de l'égalité des résultats, même sur de gros cas
 - ne permet pas de mesurer les écarts entre deux exécutions
- **STDENV_VERIF** : mécanisme *Arcane* de comparaison des valeurs des résultats, qui permet de comparer les valeurs entre deux exécutions
 - + permet de mesurer les écarts entre deux exécutions
 - produit beaucoup de données, donc limité à des cas modestes

Validation du portage

Pour valider le portage en cours

Analyse des transferts CPU/GPU :

- **nsys** (Nvidia Nsight Systems) : outil d'analyse des performances permettant (entre autres) de visualiser les transferts CPU/GPU
⇒ permet de repérer les méthodes qui impliquent des transferts





3 ■

Exemple de portage : méthode de Newton pour les équations d'état

Exemple de portage

- Présentation du portage d'un algorithme de Newton (réalisé fin d'année 2024)
 - S'articule autour de 3 méthodes, chacune ayant nécessité la mise en place de techniques de portage spécifiques :
 - Réorganisation des structures du code
 - Filtrage « Acc »
 - Suppression d'éléments de vecteurs
 - En point commun : adaptation des types de données de travail (UniqueArray, Span)
-  But : donner une illustration d'algorithmes et structures de données compatibles Acc

Newton : présentation générale de l'algorithme

```

1 template <typename T> void EquationOfStateAccService::_newtonSolve(/* ... */ FunctionForEosCall<T>& compute_function) {
2 #if not defined(PROF_ACC_DISABLED)
3   prof_acc_begin("EOSAccService::_newtonSolve");
4 #endif
5   Integer newton_iter(0); // Nombre d'itérations du Newton
6   remaining_item = true; // Indicateur de fin de convergence
7   // Indicateur de convergence pour une variable de travail / un item
8   std::vector<std::vector<Integer>> converged_item(nb_var, std::vector<Integer>(nb_elem, 0));
9
10  // Boucle du Newton
11  for ( ; newton_iter < nb_max_iteration && remaining_item ; ++newton_iter) {
12    // Appel aux équations d'état sur les matcell non convergés
13    compute_function(/* ... */);
14
15    nb_elem = newton_var[0]->raw_data.size(); // met à jour le nombre d'éléments restants
16
17    // Calcul de la convergence
18    _newtonConvergenceComputation(/* ... */);
19
20    // Détection des éléments convergés
21    _newtonDetectConvergedItems(/* ... */);
22
23    // Suppression des éléments convergés
24    _newtonRemoveConvergedItems(/* ... */);
25
26    // On compte s'il reste des éléments non convergés
27    remaining_item = (computation_group->size() != 0);
28  }
29  // Fin boucle du Newton
30 #if not defined(PROF_ACC_DISABLED)
31   prof_acc_end("EOSAccService::_newtonSolve");
32 #endif
33 }
34 }
```

1

Calcule la convergence pour chaque variable dans chaque maille

2

Déetecte les mailles où toutes les variables ont convergé

3

Enlève les mailles convergées



1 Portage du calcul de la convergence

Algorithme initial

```

1 // On définit ici la std::function comme une lambda_fonction, utilisée plus bas dans l'algorithme
2 std::function<void(const Real&, Real&, Integer&)> test_and_increment;
3
4 if (getTestBeforeIncrement()) {
5     // on teste puis on incrémenté
6     test_and_increment = [newton_tol, newton_accuracy]{
7         const Real& inner_increment, Real& newton_var_ii, Integer& converged_item_ii
8     } {
9         // Teste la convergence
10        if (math::abs(inner_increment) <= newton_tol * math::abs(newton_var_ii) + newton_accuracy) {
11            converged_item_ii = 1;
12        }
13        // Puis incrémenté la variable du Newton
14        if (converged_item_ii != 1) {
15            newton_var_ii += inner_increment;
16        }
17    };
18 } else {
19     // on incrémenté puis on teste
20     test_and_increment = [newton_tol, newton_accuracy]{
21         const Real& inner_increment, Real& newton_var_ii, Integer& converged_item_ii
22     } {
23         // Incrémente la variable
24         newton_var_ii += inner_increment;
25         // Teste la convergence
26         if (math::abs(inner_increment) <= newton_tol * math::abs(newton_var_ii) + newton_accuracy) {
27             converged_item_ii = 1;
28         }
29     };
30 } // Fin définition de la fonction locale test_and_increment
31
32 // Calcul de la convergence
33 _newtonConvergenceComputation(test_and_increment /* ... */);

```

Difficultés rencontrées :

- `std::function` et `lambda`
- `std::vector<std::vector<Integer>>`
- `std::vector` de struct contenant un `std::vector`

```

1 void EquationOfStateAccService::_newtonConvergenceComputation(
2     std::vector<std::vector<Integer>>& converged_item,
3     std::function<void(const Real&, Real&, Integer&)> test_and_increment
4 )
5 {
6     Real increment(0.); // Incrémentation du Newton
7     Integer nb_elem(newton_var[0]->raw_data.size()); // Nombre d'éléments à résoudre
8     Integer nb_var(newton_var.size()); // Nombre de variables
9
10    for (Integer ii(0) ; ii < nb_elem ; ++ii) {
11        for (Integer ivar(0) ; ivar < nb_var ; ++ivar) {
12            if (converged_item[ivar][ii] == 0) {
13                if (newton_derivative[ivar]->raw_data[ii] != 0.) {
14                    // Calcul de l'incrément
15                    Real delta_f[newton_f[ivar]->raw_data[ii]] - [newton_f_init[ivar].raw_data[ii]];
16                    increment = - delta_f / newton_derivative[ivar]->raw_data[ii];
17
18                    // Incrémentation de la variable_puis_test de CV (défaut), ou l'inverse
19                    test_and_increment(increment, [newton_var[ivar]->raw_data[ii]], converged_item[ivar][ii]);
20                }
21            }
22        }
23    }
24 }

```



1 Portage du calcul de la convergence

Solution implémentée

- std::function et lambda

```
1 template<typename FuncType>
2 void EquationOfStateAccService::_newtonConvergenceComputation(
3     FuncType test_and_increment
4 )
5 { /* ... */ }
```

```
1 if (getTestBeforeIncrement()) {
2     // on teste puis on incrémente les mailles non CV
3     auto test_and_increment = [newton_tol, newton_accuracy] ARCCORE_HOST_DEVICE [
4         const Real& inner_increment,
5         Real& newton_var_ii,
6         bool& converged_item_ii
7     ] {
8         // Test de convergence
9         if (math::abs(inner_increment) <= newton_tol * math::abs(newton_var_ii) + newton_accuracy) {
10             // L'élément est convergé
11             converged_item_ii = true;
12         }
13         // Incrémentation de la variable sur les mailles non convergées
14         if (!converged_item_ii) {
15             newton_var_ii += inner_increment;
16         }
17     };
18     _newtonConvergenceComputation<decltype(test_and_increment)>(
19         /* ... */
20         test_and_increment
21     );
22 } else {
23     // on incrémente puis on teste (méthode "historique")
24     auto test_and_increment = [newton_tol, newton_accuracy] ARCCORE_HOST_DEVICE [
25         const Real& inner_increment,
26         Real& newton_var_ii,
27         bool& converged_item_ii
28     ] {
29         // Incrémentation de la variable
30         newton_var_ii += inner_increment;
31
32         // Test de convergence
33         if (math::abs(inner_increment) <= newton_tol * math::abs(newton_var_ii) + newton_accuracy) {
34             // L'élément est convergé
35             converged_item_ii = true;
36         }
37     };
38     _newtonConvergenceComputation<decltype(test_and_increment)>(
39         /* ... */
40         test_and_increment
41     );
42 }
```

1 Portage du calcul de la convergence

Solution implémentée

- `std::vector<std::vector<Integer>>`

- `std::vector` de struct contenant un `std::vector`

- Portage de la boucle « interne »

```

1 UniqueArray<bool> m_converged_item;
2
3 template<typename FuncType>
4 void EquationOfStateAccService::_newtonConvergenceComputation(
5     FuncType test_and_increment
6 )
7 {
8     Real increment(0.); // Incrémentation du Newton
9     Integer nb_elem(newton_var[0]->raw_data.size()); // Nombre d'éléments à résoudre
10    Integer nb_var(newton_var.size()); // Nombre de variables
11
12    // Vue sur m_converged_item
13    MDSpan<bool, MDDim2> converged_item_span(m_converged_item.data(), {nb_var, nb_elem});
14
15    for (Integer ivar(0) ; ivar < nb_var ; ++ivar) {
16        auto newton_var_span(newton_var[ivar]->raw_data.span());
17        auto newton_derivative_span(newton_derivative[ivar]->raw_data.span());
18
19        Runner* runner = Acc_env::getGlobalDefaultRunner();
20        auto queue = ax::makeQueue(runner);
21        auto command = ax::makeCommand(queue);
22
23        command << RUNCOMMAND_LOOP1(iter, nb_elem) {
24            auto [ii] = iter();
25            if (!converged_item_span(ivar, ii)) {
26                if (newton_derivative_span[ii] != 0.) {
27                    // Calcul de l'incrément
28                    Real delta_f(newton_f_span[ii] - newton_f_init_span[ii]);
29                    increment = - delta_f / newton_derivative_span[ii];
30
31                    // Incrémentation de la variable puis test de CV (défaut), ou l'inverse
32                    test_and_increment(increment, newton_var_span[ii], converged_item_span(ivar, ii));
33                }
34            }
35        };
36    }
37 }
```

2 Portage de la détection des éléments convergés

Algorithme initial

Difficultés rencontrées :

- construction séquentielle des indices dans le tableau nonConvergedItemIndices

```

1 // converged_item = 0  => pas convergé
2 //           = 1  => convergé mais pas encore recopié
3 //           = -1 => convergé et recopié
4
5
6 // Détection des éléments convergés
7 UniqueArray<Integer> nonConvergedItemIndices(nb_elem);
8 int iNonConverged(0);
9 for (Integer ielem(0) ; ielem < nb_elem ; ++ielem) {
10    bool every_component_converged(true);
11    for (Integer isystemvar(0) ; isystemvar < nb_var ; ++isystemvar) {
12        if (converged_item[isystemvar][ielem] == 1) {
13            for (Integer ivar(0) ; ivar < forced_variable_temp.size() ; ++ivar) {
14                // l'élément est convergé, on le recopie dans les forced_variable initiales
15                forced_variable[ivar]->raw_data[forced_var_lookup_table[ielem]] =
16                    forced_variable_temp[ivar]->raw_data[ielem];
17            }
18            // On positionne l'indicateur à -1 pour éviter de copier des données plusieurs fois
19            converged_item[isystemvar][ielem] = -1;
20        }
21        every_component_converged = every_component_converged && (converged_item[isystemvar][ielem] == -1);
22    }
23    if (!every_component_converged) {
24        nonConvergedItemIndices[iNonConverged] = ielem;
25        iNonConverged++;
26    }
27 }
28 nonConvergedItemIndices.resize(iNonConverged);

```

2 Portage de la détection des éléments convergés

Solution implémentée

- Recopie des éléments convergés : portage « facile »

```
54 // Recopie l'élément convergé dans les forced_variable initiales
55 auto forced_variable_temp_size(forced_variable_temp.size());
56 Span<Integer> forced_var_lookup_table_view(forced_var_lookup_table.span());
57 for (Integer ivar(0) ; ivar < forced_variable_temp_size ; ++ivar) {
58     Span<Real> forced_variable_ivar(forced_variable[ivar]->raw_data.span());
59     Span<Real> forced_variable_temp_ivar(forced_variable_temp[ivar]->raw_data.span());
60
61     auto command = ax::makeCommand(rqueue_async_ptr);
62     command << RUNCOMMAND_LOOP1(iter, nConverged){
63         auto [i_elem] = iter();
64         auto index(convergedItemIndicesView[i_elem]);
65         forced_variable_ivar[forced_var_lookup_table_view[index]] = forced_variable_temp_ivar[index];
66     };
67 }
```

- Comment construire la liste des indices des éléments convergés ?

2 Portage de la détection des éléments convergés

Solution implémentée

- Comment porter l'algorithme de création de la liste des indices convergés ?
 - Algorithme de filtrage : GenericFilterer, qui permet de filtrer les éléments d'un tableau

```

24 // Filtres
25 auto nonConvergedItemIndicesView(m_nonConvergedItemIndices.span());
26 gen_filterer_inst->applyWithIndex(nb_elem,
27 [=] ARCCORE_HOST_DEVICE (Integer i_elem) -> bool
28 {
29     every_component_converged_span(i_elem) = true;
30     for(Integer isystemvar(0) ; isystemvar < nb_var ; ++isystemvar){
31         every_component_converged_span(i_elem) = every_component_converged_span(i_elem) && converged_item_span(isystemvar, i_elem);
32     }
33     return !every_component_converged_span(i_elem);
34 },
35 [=] ARCCORE_HOST_DEVICE (Integer i_elem, Integer iNonConverged) -> void {
36     nonConvergedItemIndicesView[iNonConverged] = i_elem;
37 }
38 );
39 Integer nNonConverged = gen_filterer_inst->nbOutputElement(); // induit un appel à rqueue_async_ptr->barrier()
40 m_nonConvergedItemIndices.resize(nNonConverged);
41
42 auto convergedItemIndicesView(m_convergedItemIndices.span());
43 gen_filterer_inst->applyWithIndex(nb_elem,
44 [=] ARCCORE_HOST_DEVICE (Integer i_elem) -> bool {
45     return every_component_converged_span(i_elem);
46 },
47 [=] ARCCORE_HOST_DEVICE (Integer i_elem, Integer iConverged) -> void {
48     convergedItemIndicesView[iConverged] = i_elem;
49 }
50 );
51 Integer nConverged = gen_filterer_inst->nbOutputElement(); // induit un appel à rqueue_async_ptr->barrier()

```

sélection des indices

setter

2 Portage de la détection des éléments convergés

Solution implémentée

Volume de code...

```

1 // Détection des éléments convergés
2 UniqueArray<Integer> nonConvergedItemIndices(nb_elem);
3 int iNonConverged(0);
4 for (Integer ielem(0) ; ielem < nb_elem ; ++ielem) {
5     bool every_component_converged(true);
6     for (Integer isystemvar(0) ; isystemvar < nb_var ; ++isystemvar) {
7         if (converged_item[isystemvar][ielem] == 1) {
8             for (Integer ivar(0) ; ivar < forced_variable_temp.size() ; ++ivar) {
9                 // L'élément est convergé, on le recopie dans les forced_variable initiales
10                forced_variable[ivar]->raw_data[forced_var_lookup_table[ielem]] =
11                    forced_variable_temp[ivar]->raw_data[ielem];
12            }
13            // On positionne l'indicateur à -1 pour éviter de copier des données plusieurs fois
14            converged_item[isystemvar][ielem] = -1;
15        }
16        every_component_converged = every_component_converged && (converged_item[isystemvar][ielem] == -1);
17    }
18    if (!every_component_converged) {
19        nonConvergedItemIndices[iNonConverged] = ielem;
20        iNonConverged++;
21    }
22 }
nonConvergedItemIndices.resize(iNonConverged);

```



```

1 void EquationOfStateAccService::_newtonDetectConvergedItems/* ... */
2 {
3     m_nonConvergedItemIndices.resizeNoInit(nb_elem);
4     m_convergedItemIndices.resizeNoInit(nb_elem);
5
6     // Vue sur m_converged_item
7     MDSpan<bool, MDDim2> converged_item_span(m_converged_item.data(), {nb_var, nb_elem});
8
9     auto rqueue_async_ptr, gen_filterer_inst = m_aqueue_gen_filterer.asyncRunQueueAndGenericFiltererPtr(subDomain());
10
11    // Parcours des variables forcées
12    m_every_component_converged.resizeNoInit(nb_elem);
13    auto every_component_converged_span(m_every_component_converged.span());
14
15    {
16        auto command = ax::makeCommand(rqueue_async_ptr);
17        command << RUNCOMMAND_LOOP(iter, nb_elem){
18            auto [i_elem] = iter();
19            every_component_converged_span[i_elem] = false;
20        };
21    }
22    rqueue_async_ptr->barrier();
23
24    // Filtres
25    auto nonConvergedItemIndicesView(m_nonConvergedItemIndices.span());
26    gen_filterer_inst->applyWithIndex(nb_elem,
27        [=] ARCCORE_HOST_DEVICE (Integer i_elem) -> bool {
28            every_component_converged_span[i_elem] = true;
29            for (Integer isystemvar(0) ; isystemvar < nb_var ; ++isystemvar){
30                every_component_converged_span[i_elem] = every_component_converged_span[i_elem] && converged_item_span(isystemvar, i_elem);
31            }
32            return !every_component_converged_span[i_elem];
33        },
34        [=] ARCCORE_HOST_DEVICE (Integer i_elem, Integer iNonConverged) -> void {
35            nonConvergedItemIndicesView[iNonConverged] = i_elem;
36        }
37    );
38
39    Integer nNonConverged = gen_filterer_inst->nbOutputElement(); // induit un appel à rqueue_async_ptr->barrier()
40    m_nonConvergedItemIndices.resize(nNonConverged);
41
42    auto convergedItemIndicesView(m_convergedItemIndices.span());
43    gen_filterer_inst->applyWithIndex(nb_elem,
44        [=] ARCCORE_HOST_DEVICE (Integer i_elem) -> bool {
45            return every_component_converged_span[i_elem];
46        },
47        [=] ARCCORE_HOST_DEVICE (Integer i_elem, Integer iConverged) -> void {
48            convergedItemIndicesView[iConverged] = i_elem;
49        }
50    );
51    Integer nConverged = gen_filterer_inst->nbOutputElement(); // induit un appel à rqueue_async_ptr->barrier()
52    m_convergedItemIndices.resize(nConverged);
53
54    // Recopie l'élément convergé dans les forced_variable initiales
55    auto forced_variable_temp_size=forced_variable_temp.size();
56    Span<Integer> forced_var_lookup_table_view(forced_var_lookup_table.span());
57    for (Integer ivar(0) ; ivar < forced_variable_temp_size ; ++ivar) {
58        Span<Real> forced_variable_ivar(forced_variable[ivar]->raw_data.span());
59        Span<Real> forced_variable_temp_ivar(forced_variable_temp[ivar]->raw_data.span());
60
61        auto command = ax::makeCommand(rqueue_async_ptr);
62        command << RUNCOMMAND_LOOP(iter, nConverged){
63            auto [i_elem] = iter();
64            auto index(convergedItemIndicesView[i_elem]);
65            forced_variable_ivar[forced_var_lookup_table_view[index]] = forced_variable_temp_ivar[index];
66        };
67    }
68    rqueue_async_ptr->barrier();
69 }

```

3 Portage de la suppression des éléments convergés

Algorithme initial

- Appels à la méthode `resizeFromMask`, qui élimine du tableau les indices pour lesquels `to_erase` est vrai

```

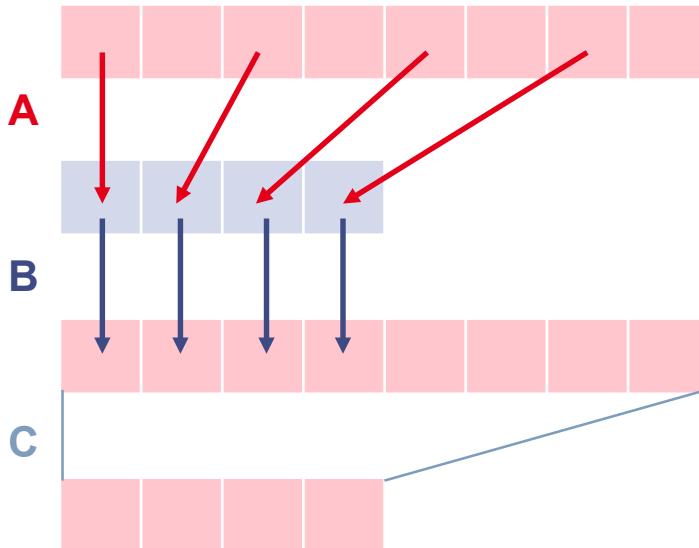
1 // Construction du tableau des éléments à supprimer
2 UniqueArray<bool> to_erase(nb_elem, false);
3 for (Integer ielem(0) ; ielem < nb_elem ; ielem++) {
4     bool every_component_converged(true);
5     for (Integer isystemvar(0) ; isystemvar < nb_var ; ++isystemvar) {
6         every_component_converged = every_component_converged && (converged_item[isystemvar][ielem] == -1);
7     }
8     if (every_component_converged) {
9         to_erase[ielem] = true;
10    }
11 }
12
13 // Variables "vector"
14 computation_group->resizeFromMask(to_erase);
15
16 // Variables "array de vector"
17 for (Integer ii(0) ; ii < newton_f_init.size() ; ++ii) {
18     newton_f_init[ii].raw_data.resizeFromMask(to_erase);
19 }
```

- Problème : intrinsèquement séquentiel !

3 Portage de la suppression des éléments convergés

Solution implémentée

- Variables « vector »



```
1 UniqueArray<T>* m_packed_computation_group_ptr;
```

```

1 // --- computation_group ---
2 m_packed_computation_group_ptr->resizeNoInit(newSize);
3 Span<T> packed_computation_group_view(m_packed_computation_group_ptr->span());
4
5 Span<T> computation_group_view(computation_group->span());
6
7 auto command0 = ax::makeCommand(queue);
8 command0 << RUNCOMMAND_LOOP1(iter, newSize) {
9     auto [i_elem] = iter();
10    packed_computation_group_view(i_elem) = computation_group_view[nonConvergedItemIndicesView[i_elem]]; A
11 };
12
13 auto command1 = ax::makeCommand(queue);
14 command1 << RUNCOMMAND_LOOP1(iter, newSize) {
15     auto [i_elem] = iter();
16    computation_group_view[i_elem] = packed_computation_group_view(i_elem);
17 };
18
19 computation_group->resizeAttributes(newSize);

```

The code snippet shows the implementation of a computation group. It starts by resizing the packed computation group without initialization. It then creates two commands: command0 and command1. Command0 uses the RUNCOMMAND_LOOP1 iterator to update elements in the packed computation group view with values from the computation group view, indexed by nonConvergedItemIndicesView[i_elem]. Command1 uses the same loop to update elements in the computation group view with values from the packed computation group view. Finally, it resizes the attributes of the computation group.

3 Portage de la suppression des éléments convergés

Solution implémentée

- Variables « array de vector »
 - Idem dans une boucle
 - MDSpan

```

1 // --- newton_f_init ---
2 auto newton_f_init_size(newton_f_init.size());
3 m_packed_newton_f_init.resizeNoInit(newton_f_init_size * newSize);
4 MDSpan<Real, MDDim2> packed_newton_f_init_view(m_packed_newton_f_init.data(), {newton_f_init_size, newSize});
5 for (Integer ii(0) ; ii < newton_f_init_size ; ++ii) {
6     Span<Real> packed_newton_f_init_ii(newton_f_init[ii].raw_data.span());
7
8     auto command0 = ax::makeCommand(queue);
9     command0 << RUNCOMMAND_LOOP1(iter, newSize) {
10        auto [i_elem] = iter();
11        packed_newton_f_init_view(ii, i_elem) = packed_newton_f_init_ii[nonConvergedItemIndicesView[i_elem]];
12    };
13
14    auto command1 = ax::makeCommand(queue);
15    command1 << RUNCOMMAND_LOOP1(iter, newSize) {
16        auto [i_elem] = iter();
17        packed_newton_f_init_ii[i_elem] = packed_newton_f_init_view(ii, i_elem);
18    };
19 }

```

- Deux commandes par variables, une seule queue

1 queue->barrier();

3 Portage de la suppression des éléments convergés

Solution implémentée

Volume de code...

```

  // Suppression des éléments convergés
  1  Utilis->resizedCompte_grace_Accrue(&nb_lines, &nb_lines);
  2  for (integer iiconom = 1; iiconom < nb_lines; iiconom++) {
  3    if (integer iiconom == 1) {
  4      for (integer iiconom = 1; iiconom < nb_lines; iiconom++) {
  5        if (integer iiconom == 1) { lystemmes[iiconom].lystemmes[iiconom] = 0;
  6        else lystemmes[iiconom].lystemmes[iiconom] = lystemmes[iiconom-1];
  7        if (iiconom > 1) lystemmes[iiconom].lystemmes[iiconom] += lystemmes[iiconom];
  8      }
  9      if (every_component_Converged)
 10        to_erase[iiconom] = true;
 11    }
 12  }
 13  // On supprime les éléments convergés dans les forced_variables et dans les pointeurs
 14  Utilis->resizedCompte_grace_Accrue(&nb_var, &nb_var);
 15  for (integer iiconom = 1; iiconom < nb_var; iiconom++) {
 16    if (forced_variable[1] == iiconom) {
 17      forced_variable[1] = forced_variable[2];
 18      forced_variable[2] = 0;
 19    }
 20  }
 21  // On supprime le niveau _1st
 22  Utilis->resizedCompte_grace_Accrue(&nb_niveau_1st, &nb_niveau_1st);
 23  for (integer iiconom = 1; iiconom < nb_niveau_1st; iiconom++) {
 24    if (niveau_1st[1] == iiconom) {
 25      niveau_1st[1] = niveau_1st[2];
 26      niveau_1st[2] = 0;
 27    }
 28  }
 29  // On supprime les racines convergées
 30  computation_group->resizedCompte_grace();
 31  // On supprime les racines convergées
 32  if (compteur_cv_treatment == NonConvergingTreatment_Diophant) {
 33    for (integer ioverd = 1; ioverd < nb_overd; ioverd++) {
 34      if (diophant_overr[1] == ioverd) {
 35        diophant_overr[1] = diophant_overr[2];
 36        diophant_overr[2] = 0;
 37        if (nb_overr == 1) nb_overr = 0;
 38      }
 39    }
 40    // Mise à zéro d'itemcv. Il faut faire une redimensionne converted_item !!!
 41    itemcv[1] = 0;
 42    itemcv[2] = 0;
 43    itemcv[3] = 0;
 44    Utilis->resizedCompte_grace_Accrue(&converted_itemcv, &converted_itemcv);
 45  }
 46
```



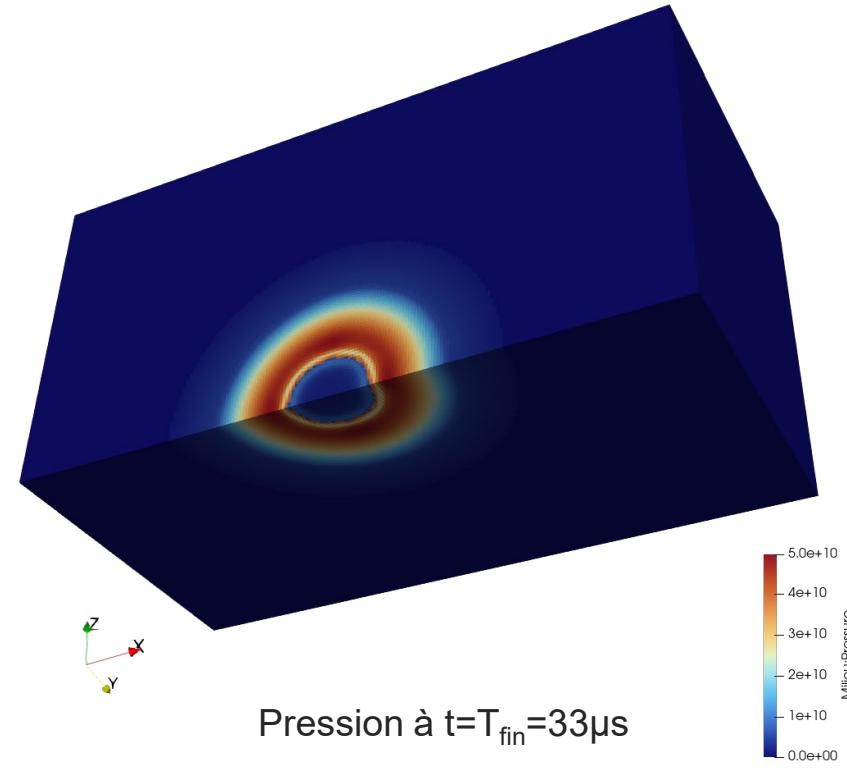
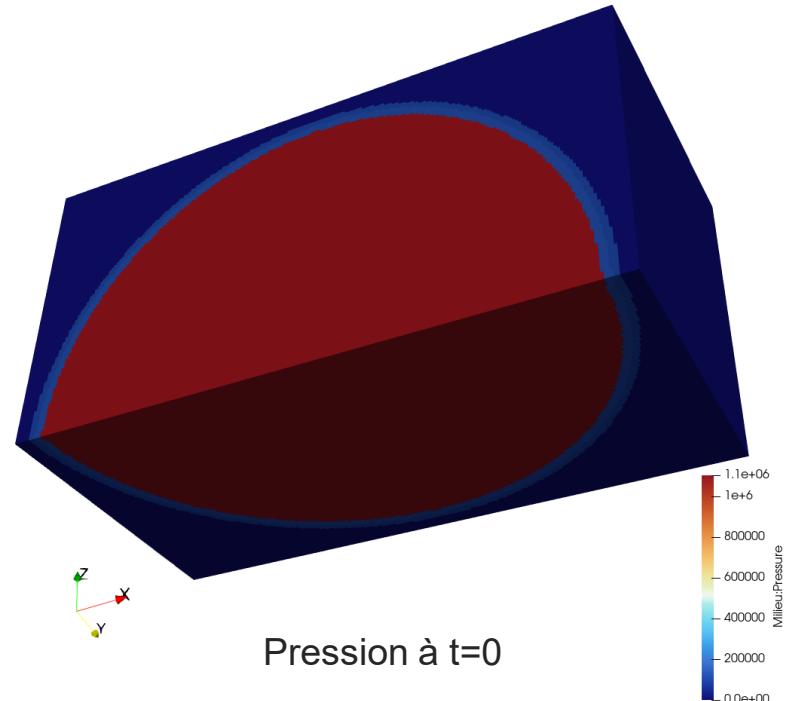


4. Performances

Cas utilisé pour étude des performances

Simulation d'implosion de coquilles contenant du gaz

- Implosion avec condition initiale de vitesse centripète
- Coquilles traitées en élasto ; gaz traité en hydro
- Cas test numériquement simple
- Cas significatifs de 3 à 12 millions de mailles



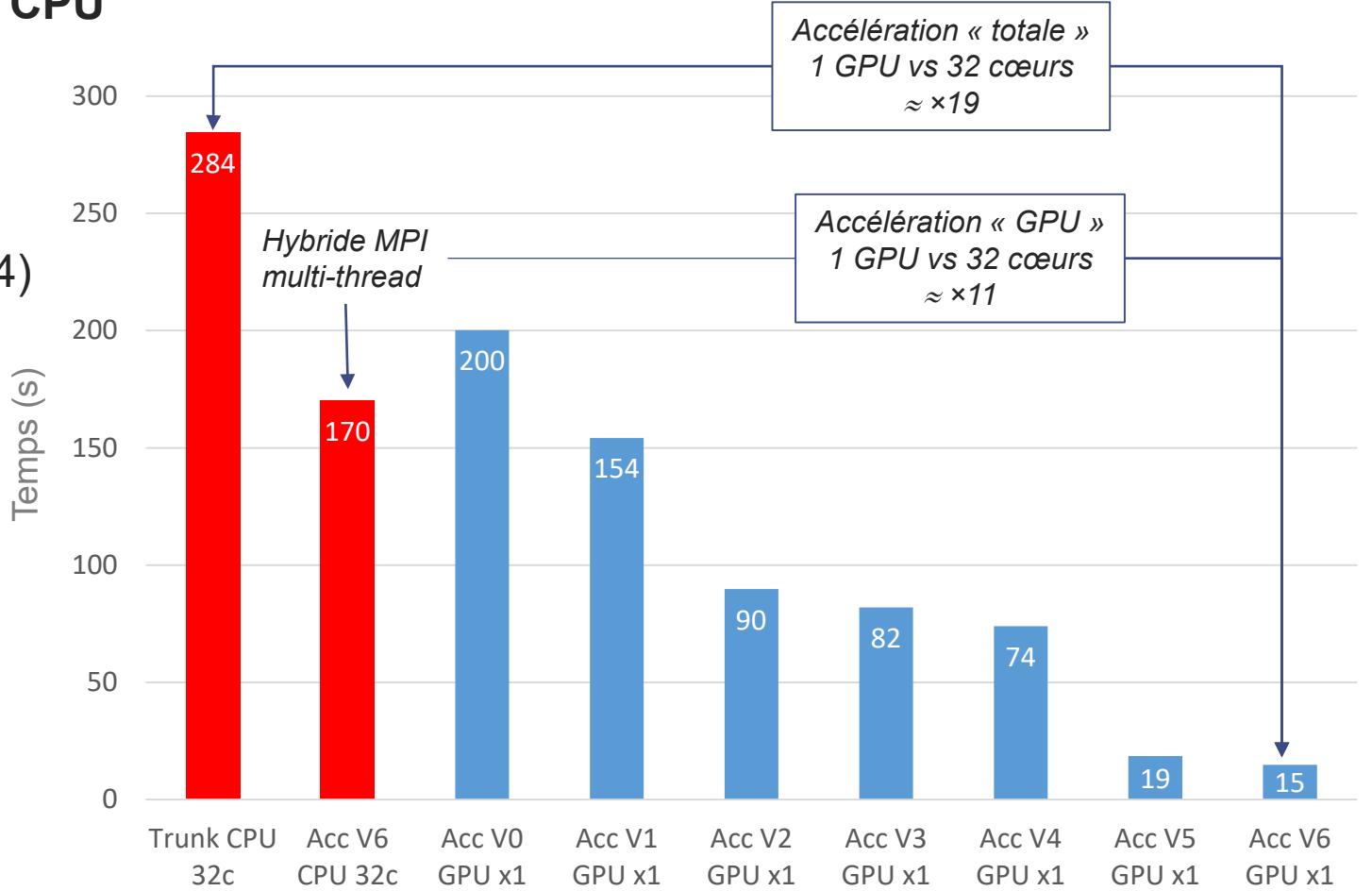
Evolution des performances

Performances mono-GPU vs 32 cœurs CPU

- Exemple à 3 millions de mailles
- Temps des 100 premières itérations
- Machine de prototypage GPU
- Versions successives du code (06/2024)

- API accélérateur → multithread

- Gains :
 - propre au GPU : ×11
 - total : ×19

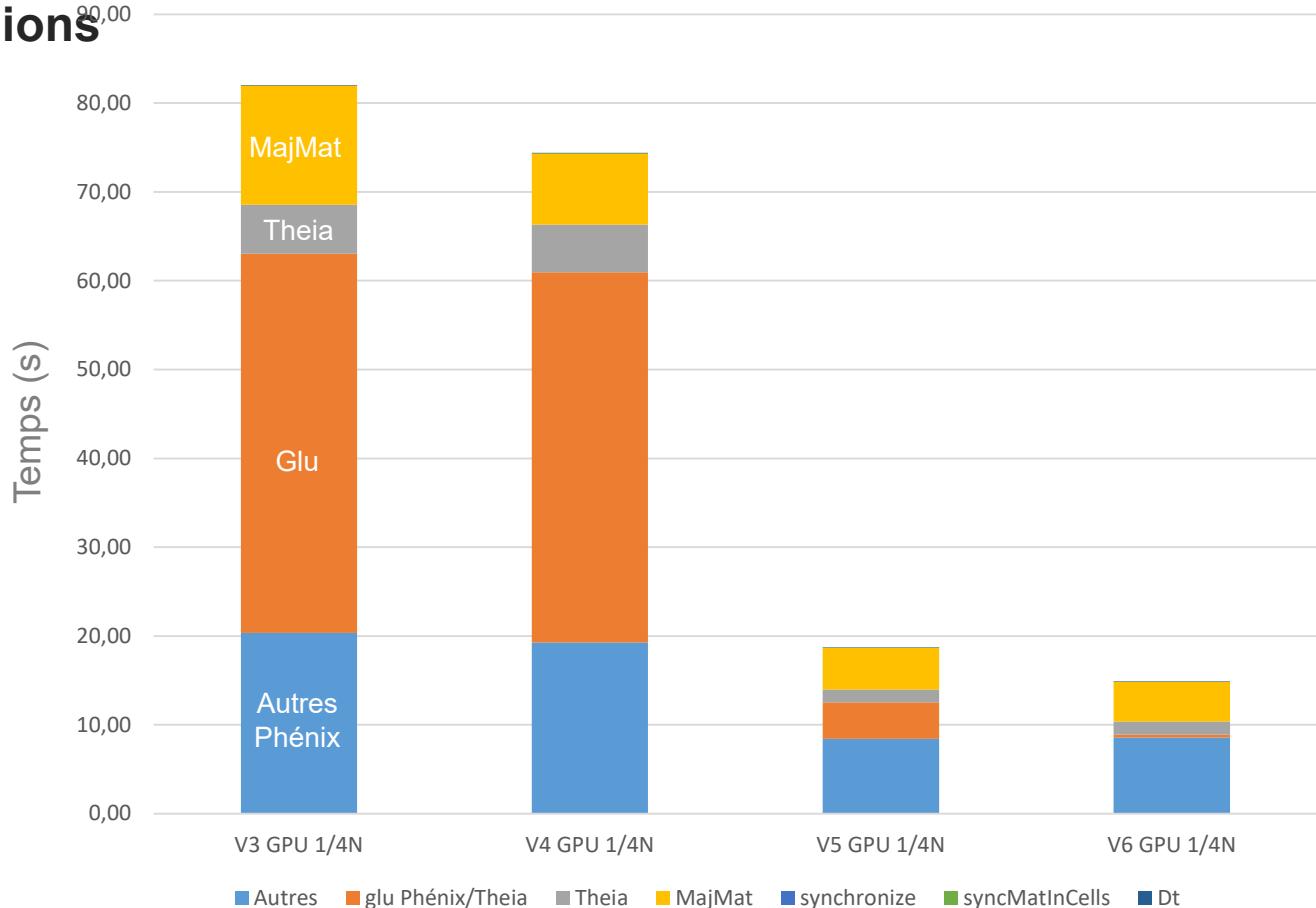


Evolution des performances

Répartition des coûts en fonction des versions

- $3 \cdot 10^6$ mailles, 100 itérations
- Performances mono-GPU
- Versions successives du code

- Améliorations significatives sur
 - Théia
 - Glu



MajMat = mise à jour var. multi-mat dans Arcane::endUpdate
Dt = calcul pas de temps + MPI_Allreduce

Evolution des performances

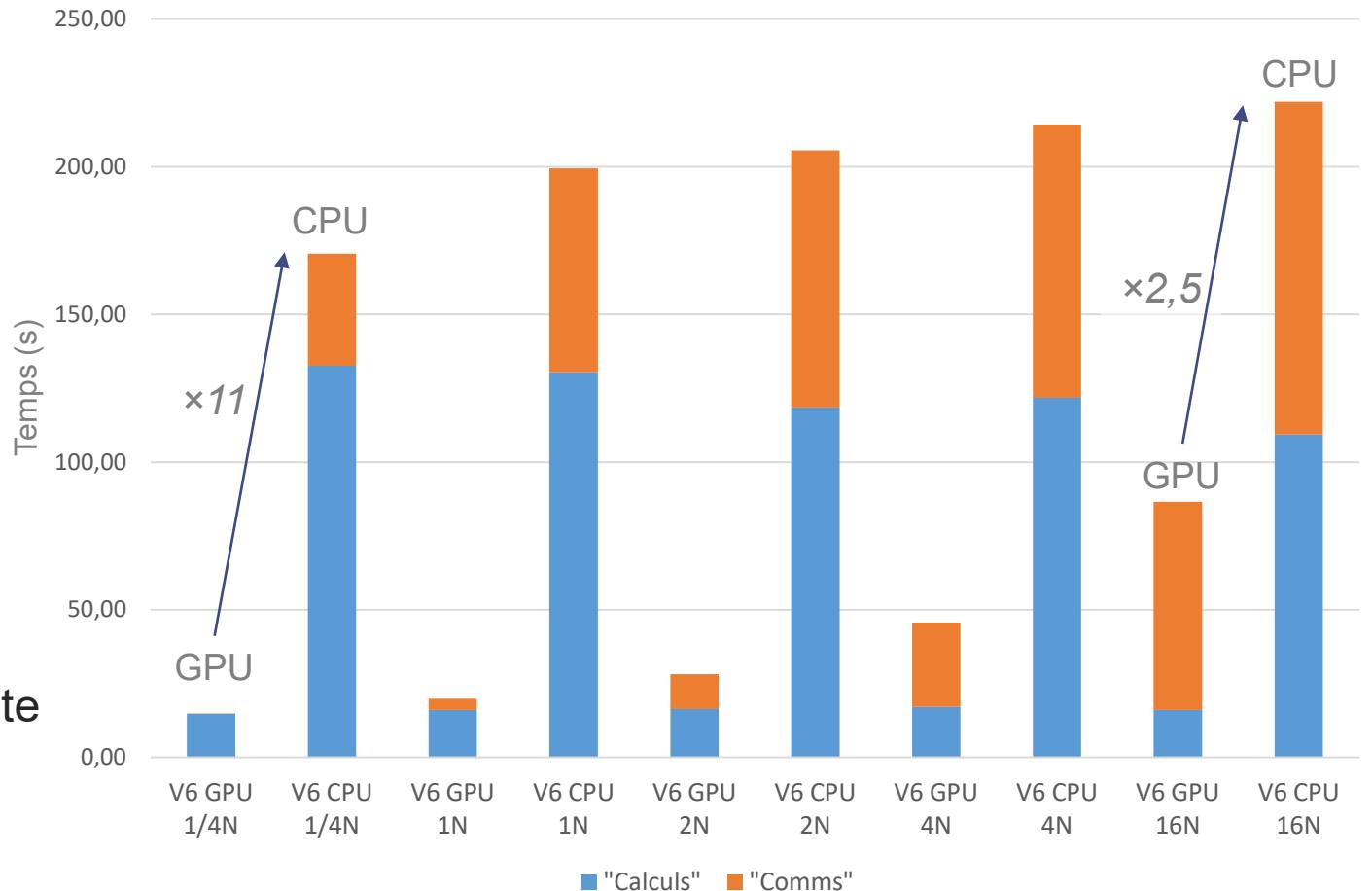
Extensibilité faible (la taille du problème croît proportionnellement aux ressources)

- 100 itérations

Nb nœuds	GPU	Cœurs CPU	Millions mailles
0,25	1	32	3,0
1	4	128	12,4
2	8	256	24,3
4	16	512	50,8
16	64	2048	194,6

- Observations :

 - Temps de calcul ~constant
 - Temps des communications augmente





5 ■ Pour conclure



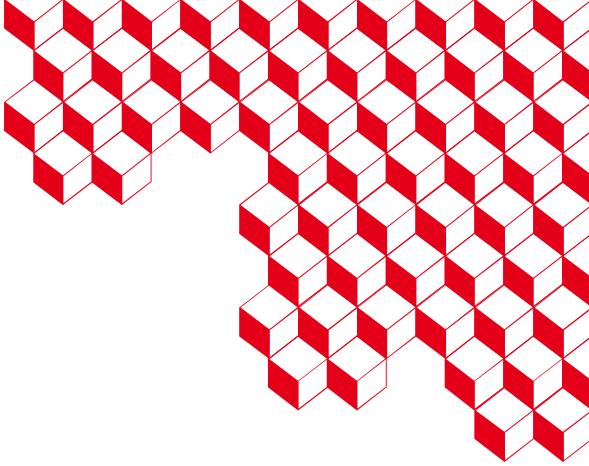
Conclusion

- Travail de longue haleine
 - Ce qui nous prend du temps : transformer les **algorithmes** et les **types de données** pour qu'ils soient portables GPU
 - Les performances ne sont obtenues qu'à la fin du portage
- L'API accélérateur [Arcane](#) nous facilite le portage
- L'équipe [Arcane](#) est très réactive pour répondre à nos demandes



Prochaines étapes

- Portage de plus en plus de modèles, pour accélérer des cas utilisateurs de plus en plus complexes
 - Difficultés à venir : configurations non rencontrées jusqu'ici, moins compatibles avec le fonctionnement GPU ?
- Mise en production de la version GPU de Phénix : beaucoup de questions soulevées
 - Comment teste-t-on que les services portés continuent de produire les mêmes résultats que les services initiaux ? (combinatoire {service classique, service Acc}×{compilation CPU, compilation GPU})
 - Fait-on disparaître les services qui n'utilisent pas l'API accélérateur ? A quelle échéance ?
 - Comment forme-t-on l'équipe de développement ?
 - Comment maintient-on le gain de performance GPU vs CPU ?



Merci

**Commissariat à l'énergie atomique et aux
énergies alternatives**
Centre DAM Ile de France Bruyères-le-Châtel
91297 Arpajon Cedex, France
simon.pomarede@cea.fr
Standard. + 33 1 69 26 40 00