Bibliothèque générique multi-cible d'algèbre linéaire

Wilfried Kirschenmann, EDF R&D - Supélec

Laurent Plagne, EDF R&D

Stéphane Vialle, Supélec - INRIA

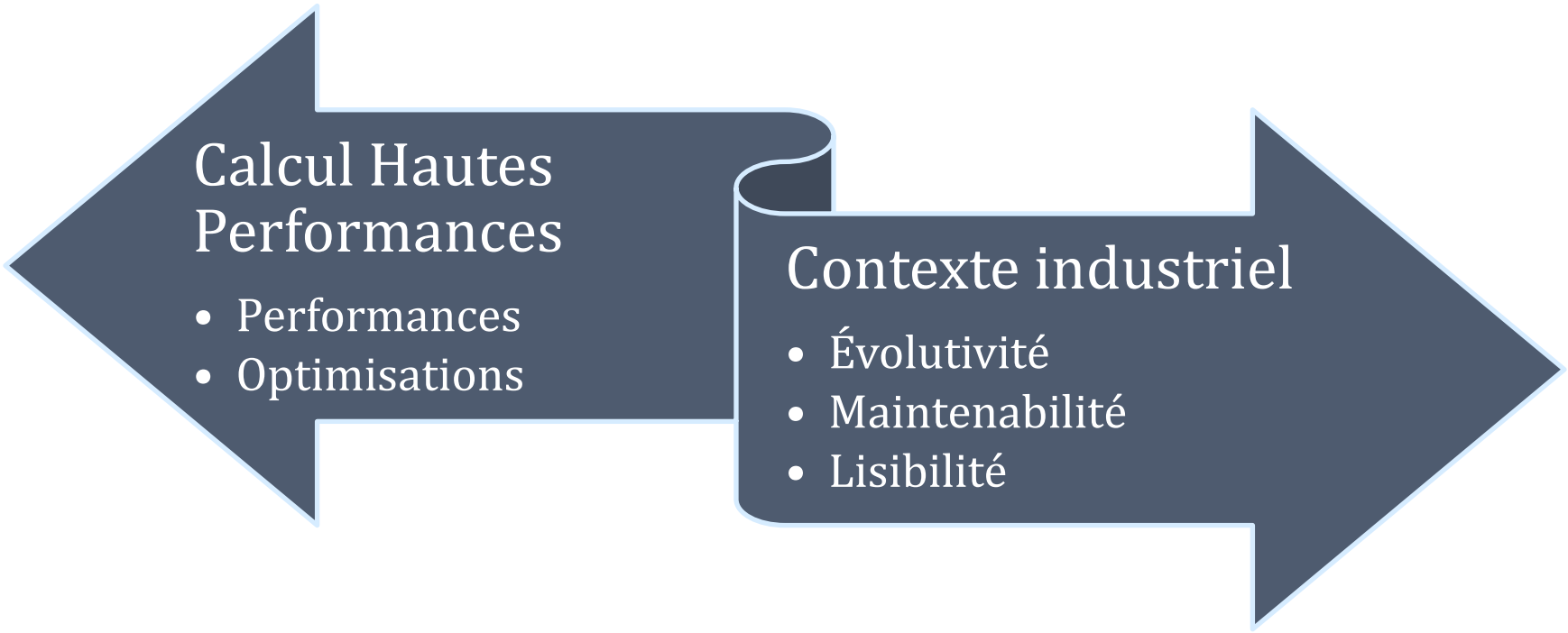
Sommaire

1. Introduction
2. DSL de l’algèbre linéaire creux structuré : Legolas++
3. Programmation de processeurs parallèles
4. Parallélisation d'un code industriel d’algèbre linéaire
5. Conclusion

1. Introduction

1. **Introduction**
   - Problématiques du HPC dans un contexte industriel
   - Différentes approches
   - Des accélérateurs de calcul
2. DSL de l'algèbre linéaire creux structuré : Legolas++
3. Programmation de processeurs parallèles
4. Parallélisation d'un code industriel d'algèbre linéaire
5. Conclusion

1. Problématiques du HPC dans un contexte industriel



1. Différentes approches

Langage

- Expressivité maximale
- Performances rarement (jamais ?) optimales

Bibliothèques

- Performances maximales
- Aucune expressivité

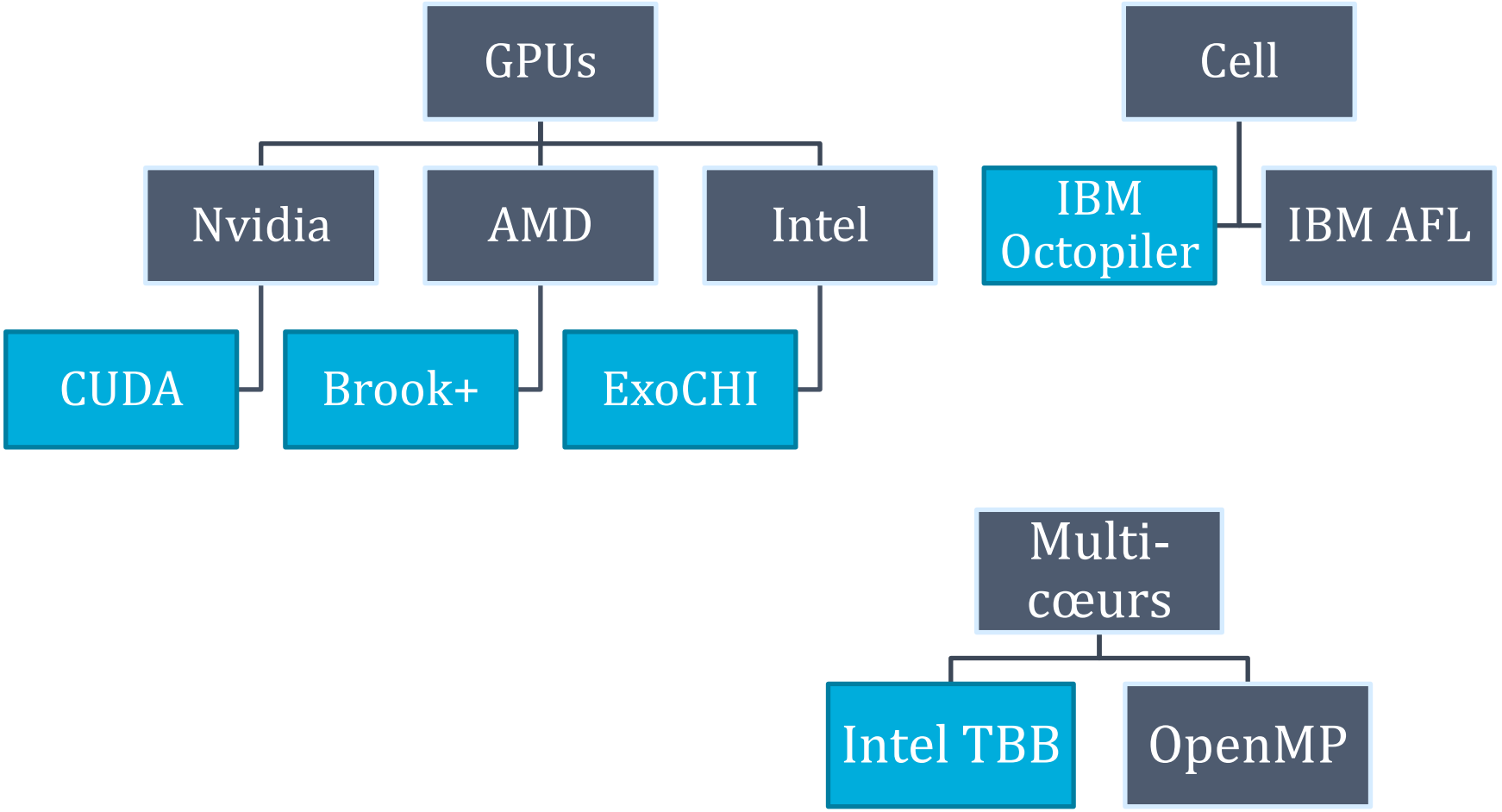
« Domain Specific Language » (DSL)

- Expressivité étendue **dans le domaine**
- Très bonnes performances **dans le domaine**

1. Différentes approches

Fonction mathématique	Langage (C)	Bibliothèques (BLAS)	DSL (Legolas++)
A+=B	add1(A, B)	cblas_saxpy(N, 1, B, 1, A, 1)	A+=B
A+=B+C	add2(A, B, C)	~~cblas_saxpy(N, 1, B, 1, A, 1)~~ ~~cblas_saxpy(N, 1, C, 1, A, 1)~~	A+=B+C

1. Des accélérateurs de calcul



2. DSL de l'algèbre linéaire creux structuré : Legolas++

1. Introduction
2. **DSL de l'algèbre linéaire creux structuré : Legolas++**
   - Possibilités
   - Exemple de code
   - Réponse à la problématique du HPC industriel
   - Performances
3. Programmation de processeurs parallèles
4. Parallélisation d'un code industriel d'algèbre linéaire
5. Conclusion

SPNMatrix=D(DSPNMatrix)+R(ScatteringMatrix)

SPNMonogroupMatrix=D(DMonoGroupMatrix)+R(RMonoGroupMatrix)

DiffusionMatrix

$$\begin{bmatrix} \hat{C}_x(\mathcal{D}_\psi^{gh}) & & -\hat{B}_x\hat{P}_{xx} \\ & \hat{C}_y(\mathcal{D}_\psi^{gh}) & -\hat{B}_y\hat{P}_{xy} \\ \hat{P}_{xx}\hat{B}_x^t & \hat{P}_{yx}\hat{B}_y^t & \hat{F}(\mathcal{D}_\phi^{gh}) \end{bmatrix}$$

H.D(0,0).D(0,0)

HarmonicCouplingMatrix

$$\begin{bmatrix} 0 & 0 & \alpha\hat{B}_x\hat{P}_{xx} \\ 0 & 0 & \alpha\hat{B}_y\hat{P}_{xy} \\ 0 & 0 & 0 \end{bmatrix}$$

H.D(0,0).R(0,1)

HarmonicCouplingMatrix

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \beta\hat{P}_{xx}\hat{B}_x^t & \beta\hat{P}_{yx}\hat{B}_y^t & 0 \end{bmatrix}$$

H.D(0,0).R(1,0)

DiffusionMatrix

$$\begin{bmatrix} \hat{C}_x(\mathcal{D}_\psi^{gh}) & & -\hat{B}_x\hat{P}_{xx} \\ & \hat{C}_y(\mathcal{D}_\psi^{gh}) & -\hat{B}_y\hat{P}_{xy} \\ \hat{P}_{xx}\hat{B}_x^t & \hat{P}_{yx}\hat{B}_y^t & \hat{F}(\mathcal{D}_\phi^{gh}) \end{bmatrix}$$

H.D(0,0).D(1,1)

H(0,0)=H.D(0,0)

SPNMonoGroupScatteringMatrix

SpaceScatteringMatrix

$$\begin{bmatrix} \hat{C}_x(\mathcal{S}_\psi^{gg'h}) & & \\ & \hat{C}_y(\mathcal{S}_\psi^{gg'h}) & \\ & & \hat{F}(\mathcal{S}_\phi^{gg'h}) \end{bmatrix}$$

H.R(0,1)(0,0)

SpaceScatteringMatrix

$$\begin{bmatrix} 0 & & \\ & 0 & \\ & & 0 \end{bmatrix}$$

H.R(0,1)(1,1)

H(0,1)=H.R(0,1)

SPNMonoGroupScatteringMatrix

SpaceScatteringMatrix

$$\begin{bmatrix} \hat{C}_x(\mathcal{S}_\psi^{gg'h}) & & \\ & \hat{C}_y(\mathcal{S}_\psi^{gg'h}) & \\ & & \hat{F}(\mathcal{S}_\phi^{gg'h}) \end{bmatrix}$$

H.R(1,0)(0,0)

SpaceScatteringMatrix

$$\begin{bmatrix} 0 & & \\ & 0 & \\ & & 0 \end{bmatrix}$$

H.R(1,0)(1,1)

H(1,0)=H.R(1,0)

SPNMonogroupMatrix=D(DMonoGroupMatrix)+R(RMonoGroupMatrix)

DiffusionMatrix

$$\begin{bmatrix} \hat{C}_x(\mathcal{D}_\psi^{gh}) & & -\hat{B}_x\hat{P}_{xx} \\ & \hat{C}_y(\mathcal{D}_\psi^{gh}) & -\hat{B}_y\hat{P}_{xy} \\ \hat{P}_{xx}\hat{B}_x^t & \hat{P}_{yx}\hat{B}_y^t & \hat{F}(\mathcal{D}_\phi^{gh}) \end{bmatrix}$$

H.D(1,1).D(0,0)

HarmonicCouplingMatrix

$$\begin{bmatrix} 0 & 0 & \alpha\hat{B}_x\hat{P}_{xx} \\ 0 & 0 & \alpha\hat{B}_y\hat{P}_{xy} \\ 0 & 0 & 0 \end{bmatrix}$$

H.D(1,1).R(0,1)

HarmonicCouplingMatrix

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \beta\hat{P}_{xx}\hat{B}_x^t & \beta\hat{P}_{yx}\hat{B}_y^t & 0 \end{bmatrix}$$

H.D(1,1).R(1,0)

DiffusionMatrix

$$\begin{bmatrix} \hat{C}_x(\mathcal{D}_\psi^{gh}) & & -\hat{B}_x\hat{P}_{xx} \\ & \hat{C}_y(\mathcal{D}_\psi^{gh}) & -\hat{B}_y\hat{P}_{xy} \\ \hat{P}_{xx}\hat{B}_x^t & \hat{P}_{yx}\hat{B}_y^t & \hat{F}(\mathcal{D}_\phi^{gh}) \end{bmatrix}$$

H.D(1,1).D(1,1)

H(1,1)=H.D(1,1)

SPNMonogroupMatrix=D(DMonoGroupMatrix)+R(RMonoGroupMatrix)

DiffusionMatrix

$$\begin{bmatrix} \hat{\mathrm{C}}_x(\mathcal{D}_\psi^{gh}) & & -\hat{\mathrm{B}}_x\hat{\mathrm{P}}_{xx} \\ & \hat{\mathrm{C}}_y(\mathcal{D}_\psi^{gh}) & -\hat{\mathrm{B}}_y\hat{\mathrm{P}}_{xy} \\ \hat{\mathrm{P}}_{xx}\hat{\mathrm{B}}_x^t & \hat{\mathrm{P}}_{yx}\hat{\mathrm{B}}_y^t & \hat{\mathrm{F}}(\mathcal{D}_\phi^{gh}) \end{bmatrix}$$

H.D(0,0).D(0,0)

HarmonicCouplingMatrix

$$\begin{bmatrix} 0 & 0 & \alpha\hat{\mathrm{B}}_x\hat{\mathrm{P}}_{xx} \\ 0 & 0 & \alpha\hat{\mathrm{B}}_y\hat{\mathrm{P}}_{xy} \\ 0 & 0 & 0 \end{bmatrix}$$

H.D(0,0).R(0,1)

HarmonicCouplingMatrix

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \beta\hat{\mathrm{P}}_{xx}\hat{\mathrm{B}}_x^t & \beta\hat{\mathrm{P}}_{yx}\hat{\mathrm{B}}_y^t & 0 \end{bmatrix}$$

H.D(0,0).R(1,0)

DiffusionMatrix

$$\begin{bmatrix} \hat{\mathrm{C}}_x(\mathcal{D}_\psi^{gh}) & & -\hat{\mathrm{B}}_x\hat{\mathrm{P}}_{xx} \\ & \hat{\mathrm{C}}_y(\mathcal{D}_\psi^{gh}) & -\hat{\mathrm{B}}_y\hat{\mathrm{P}}_{xy} \\ \hat{\mathrm{P}}_{xx}\hat{\mathrm{B}}_x^t & \hat{\mathrm{P}}_{yx}\hat{\mathrm{B}}_y^t & \hat{\mathrm{F}}(\mathcal{D}_\phi^{gh}) \end{bmatrix}$$

H.D(0,0).D(1,1)

2. Legolas++ : exemple de code

```
Typedef Legolas++ : :Vector<double> Vector;
Vector a, b, c;
…
c = 3*b;
```

```
for (int i = 0 ; i < c.size() ; i++)
    c[i] += 3*b[i];
```

```
…
a = 4*b–7*c;
```

```
for (int i = 0 ; i < a.size() ; i++)
    a[i] = 4*b[i]-7*c[i];
```

```
…
```

2. Legolas++ : réponses aux problématiques du HPC industriel

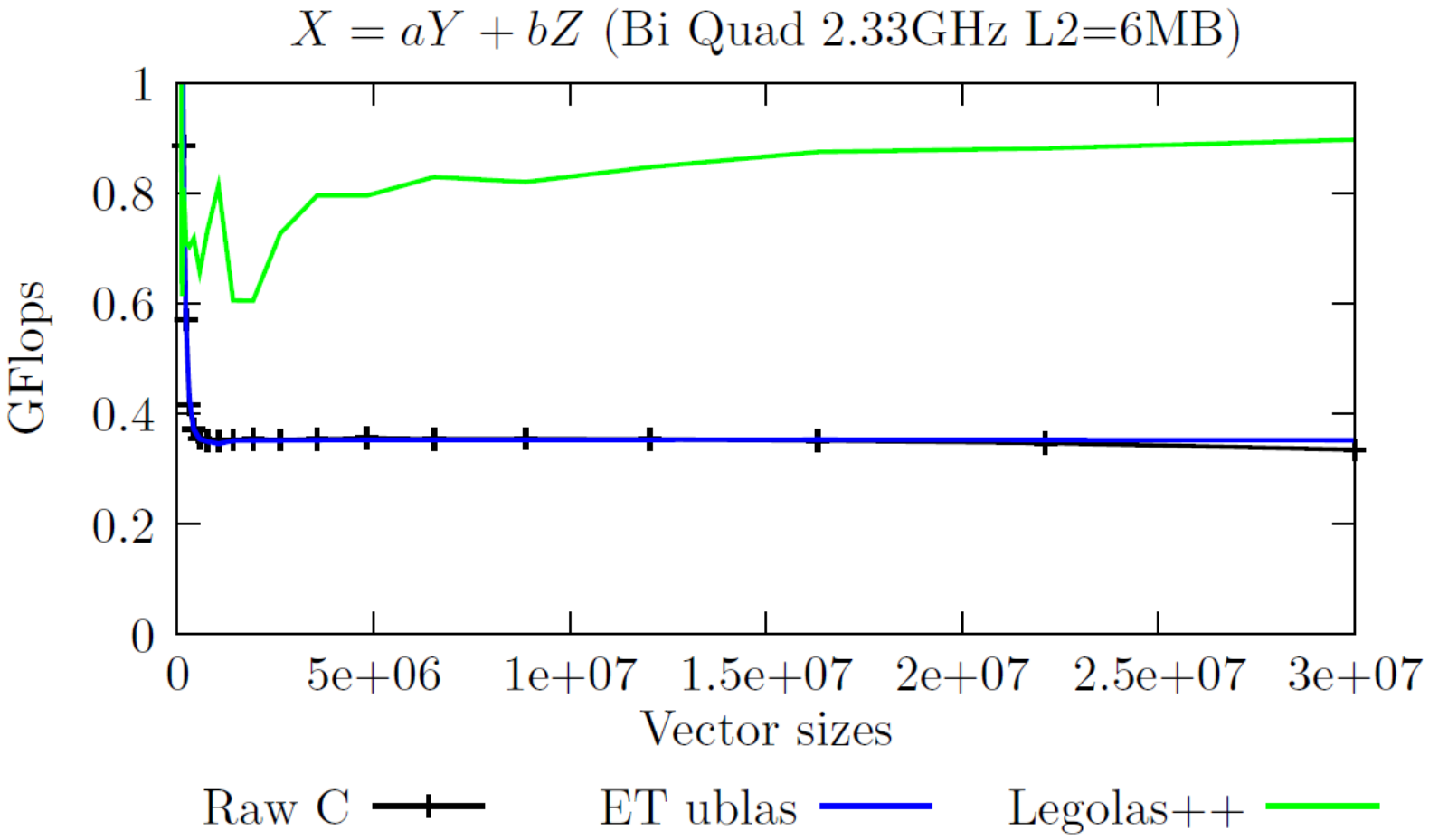
Performances

- Allocation paresseuse
- Évaluation paresseuse
- Pas de vecteurs temporaires
- Instructions optimisées en assembleur

Maintenance et évolutivité

- Écriture mathématique
- Code petit
- Optimisations localisées
- Performances garanties

2. Legolas++ : performances



3. Programmation de processeurs parallèles

1. Introduction
2. DSL de l'algèbre linéaire creux structuré : Legolas++
3. **Programmation de processeurs parallèles**
   - **Principes généraux**
   - CPU multi-cœur et Intel Threading Building Blocks
   - Les processeurs graphiques et CUDA
4. Parallélisation d'un code industriel d'algèbre linéaire
5. Conclusion

3. Programmation de processeurs parallèles : principes

Boucles que nous souhaitons paralléliser :

```
for (int i = 0 ; i < a.size() ; i++)
    f(i);
```

Avec :

f qui ne provoque pas d'effets de bord

Exemple :

```
void f ( const int & i ) {
   a[i]+=b[i]+c[i];
}
```

a, b et c étant trois tableaux de flottants

3. Programmation de processeurs parallèles : principes

```
struct Add3 {

        Add3 (Vector & a, const Vector & b, const Vector & c) :
          a_(a), b_(b), c_(c) {}

      void operator() ( const int & i ) const {
          a[i]+=b[i]+c[i];
      }

   private :
      Vector & a;
      const Vector & b;
      const Vector & c;
};
```

3. Programmation de processeurs parallèles : principes

```
…
for (int i = 0 ; i < a.size() ; i++)
    a[i] += b[i]+c[i];
…
```

```
…
const Add3 functor(a,b,c);
for (int i = 0 ; i < a.size() ; i++)
    functor(i);
…
```

3. Programmation de processeurs parallèles

1. Introduction
2. DSL de l'algèbre linéaire creux structuré : Legolas++
3. **Programmation de processeurs parallèles**
   - Principes généraux
   - **CPU multi-cœur et Intel Threading Building Blocks**
   - Les processeurs graphiques et CUDA
4. Parallélisation d'un code industriel d'algèbre linéaire
5. Conclusion

3. Programmation de processeurs parallèles : CPU multi-cœur et TBB

```
class tbb : :blocked_range<T>;
```

Sert à définir sur quel intervalle la boucle pourra être parallélisée

```
void tbb : :parallel_for<RANGE, BODY>(const RANGE&
   range, const BODY& body);
```

Fonction qui permet de paralléliser une boucle for effectuant des appels à body()

3. Programmation de processeurs parallèles : CPU multi-cœur et TBB

```
struct Add3 {
        Add3 (Vector & a, const Vector & b, const Vector & c) :
            a_(a), b_(b), c_(c) {}
        void operator() ( const int & i ) const {
            a[i]+=b[i]+c[i];
        }
        void operator() ( const tbb : :blocked_range<int> & range ) const {
            for (int i = range.begin() ; i < range.end() ; i++)
                (*this)(i);
        }
    private :
        Vector & a;
        const Vector & b;
        const Vector & c;
};
```

3. Programmation de processeurs parallèles : CPU multi-cœur et TBB

```
…
const Add3 functor(a,b,c);
for (int i = 0 ; i < a.size() ; i++)
    functor(i);
…
```

```
…
typedef tbb : :blocked_range<int> Range;
…
const Add3 functor(a,b,c);
const Range range(0, a.size());
tbb : :parallel_for< Range, Add3 >(range, functor);
…
```

3. Programmation de processeurs parallèles

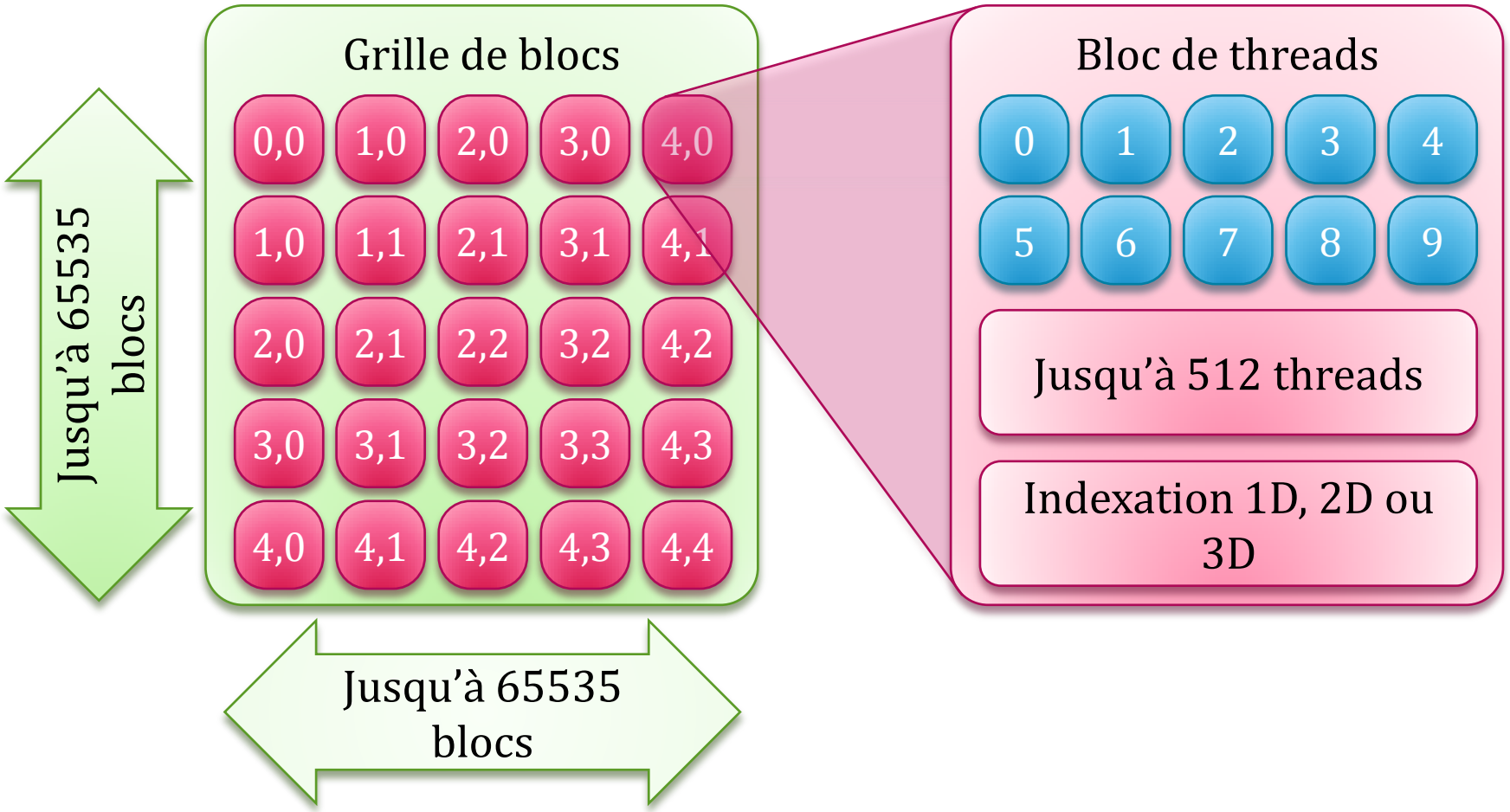
1. Introduction
2. DSL de l'algèbre linéaire creux structuré : Legolas++
3. **Programmation de processeurs parallèles**
   - Principes généraux
   - CPU multi-cœur et Intel Threading Building Blocks
   - **Les processeurs graphiques et CUDA**
4. Parallélisation d'un code industriel d'algèbre linéaire
5. Conclusion

3. Programmation de processeurs parallèles : GPUs et CUDA

Le GPU G80 (G200) en chiffres :

- Puissance de calcul théorique de 512 Gflops (1 Tflops)
- Bande passante mémoire de 74 Go/s (140 Go/s)
- 16 (30) cœurs SIMD 8 voies: 128 (240) unités de calcul
- Jusqu'à 12 288 threads simultanés (30 720)
- Jusqu'à 1.5 Go de RAM (4 Go)

3. Programmation de processeurs parallèles : GPUs et CUDA



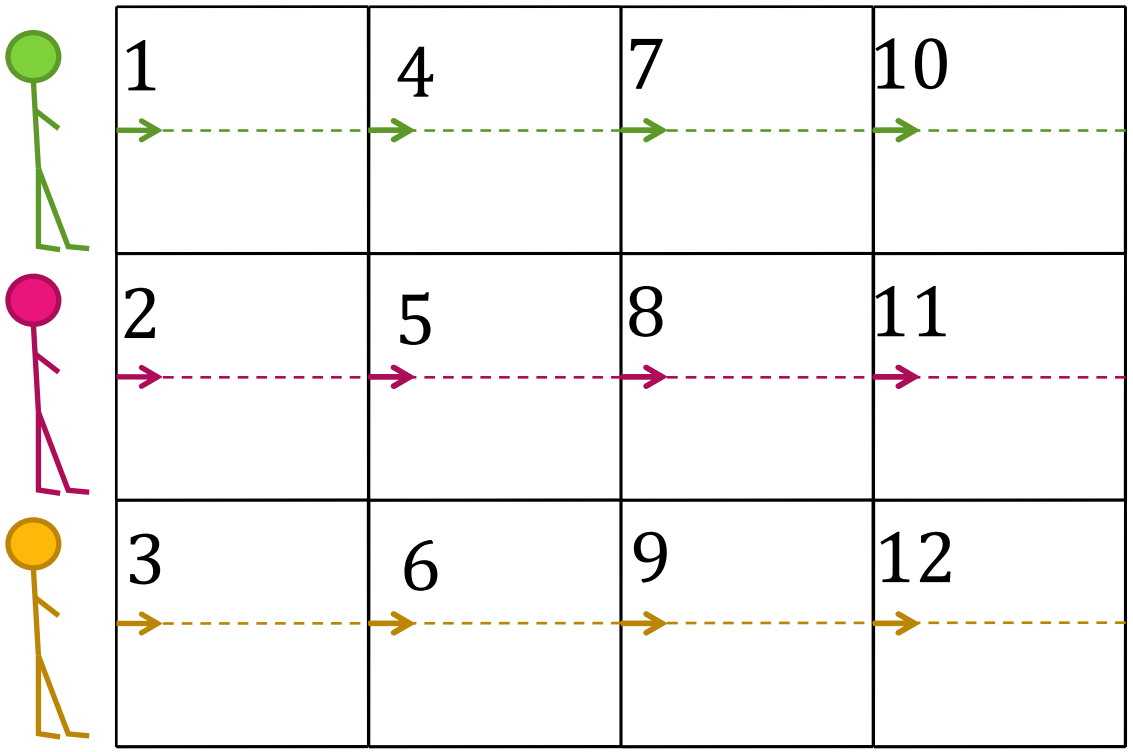
3. Programmation de processeurs parallèles : GPUs et CUDA

```
void cuda_parallel_for_Add3(const Add3 adder, const int N) {
    const int bSize = 512;
    const int nbEltsPerThreads = 4;
    int gSize = N/(bSize*nbEltsPerThreads)+1;
    while (gSize > 65535)
        gSize /=2;
    const dim3 blockDim(bSize, 1, 1);
    const dim3 gridDim(gSize, 1, 1);
    add3Kernel<<< gridDim, blockDim >>>( adder, N);
}
```

3. Programmation de processeurs parallèles : GPUs et CUDA

```
__global__ void add3Kernel(const Add3 adder, const int N) {
    const int bx = blockIdx.x;
    const int tx = threadIdx.x;
    const int bSize = blockDim.x;
    const int gSize = gridDim.x;
    const int nbThreads = bSize*gSize;
    const int threadId = tx+bx*bSize;
    for (int i = threadId ; i < N ; i += nbThreads )
        adder(i);
}
```

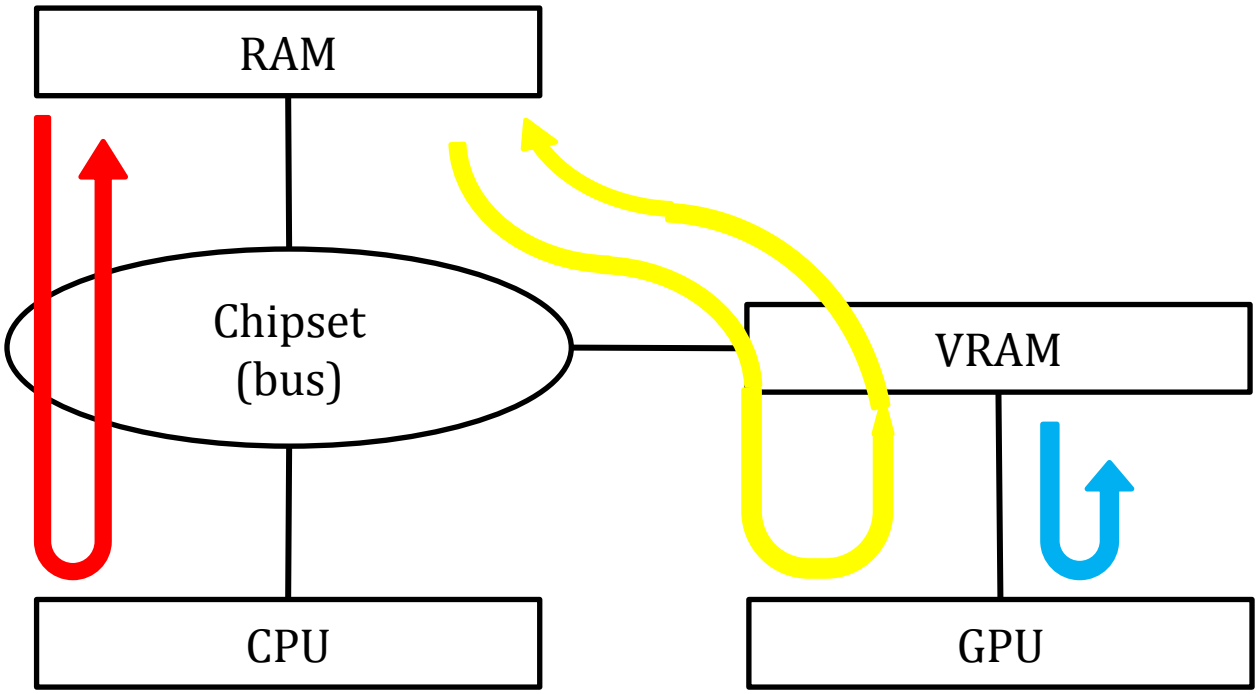
3. Programmation de processeurs parallèles : GPUs et CUDA



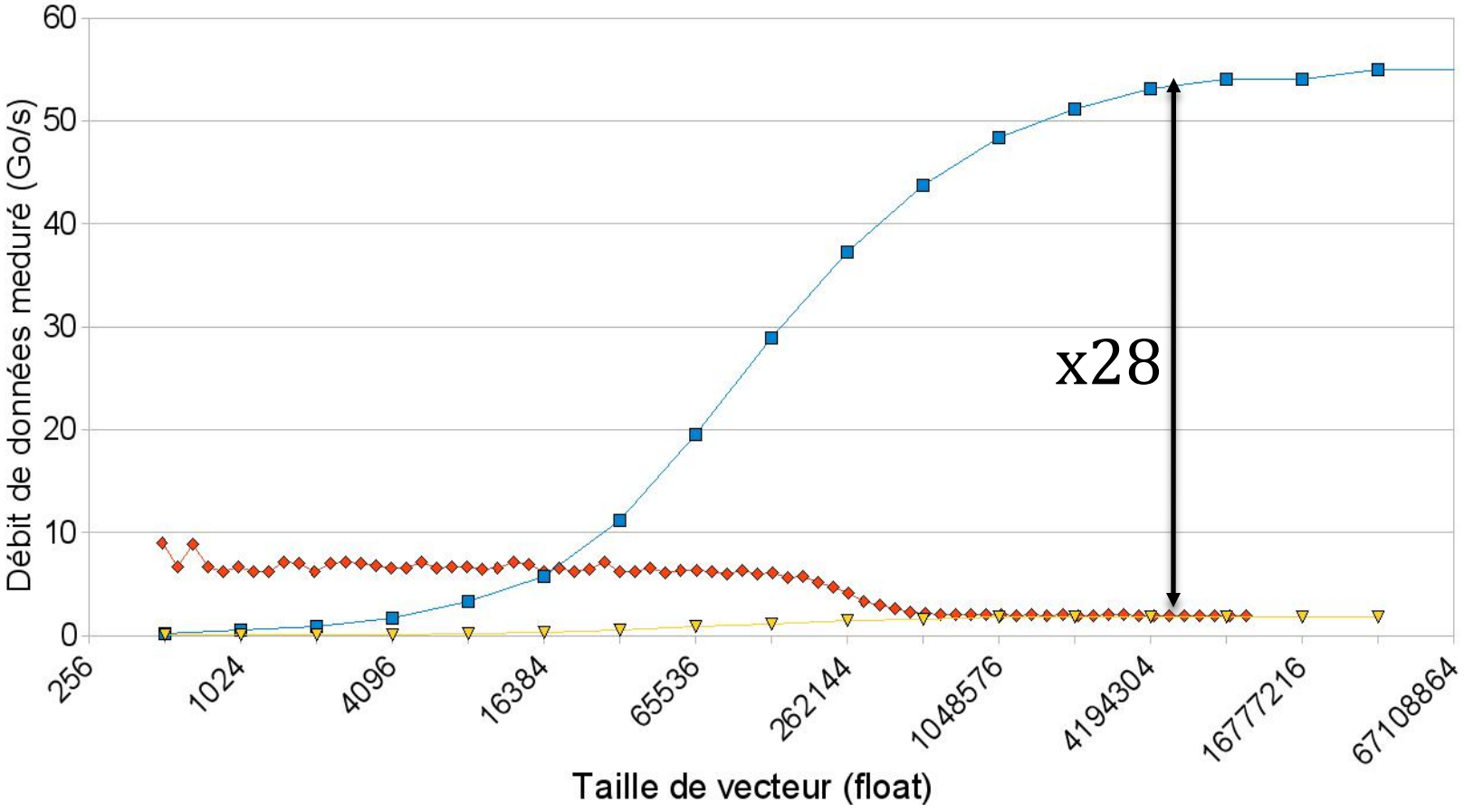
3. Programmation de processeurs parallèles : GPUs et CUDA

- Fonction testée : saxpy

  foreach i : Y[i] = a.X[i] + Y[i]



3. Programmation de processeurs parallèles : GPUs et CUDA



3. Programmation de processeurs parallèles : GPUs et CUDA

Outils de développement

- Compilateur NVCC
- CUDA profiler
- Émulateur de GPU supportant gdb

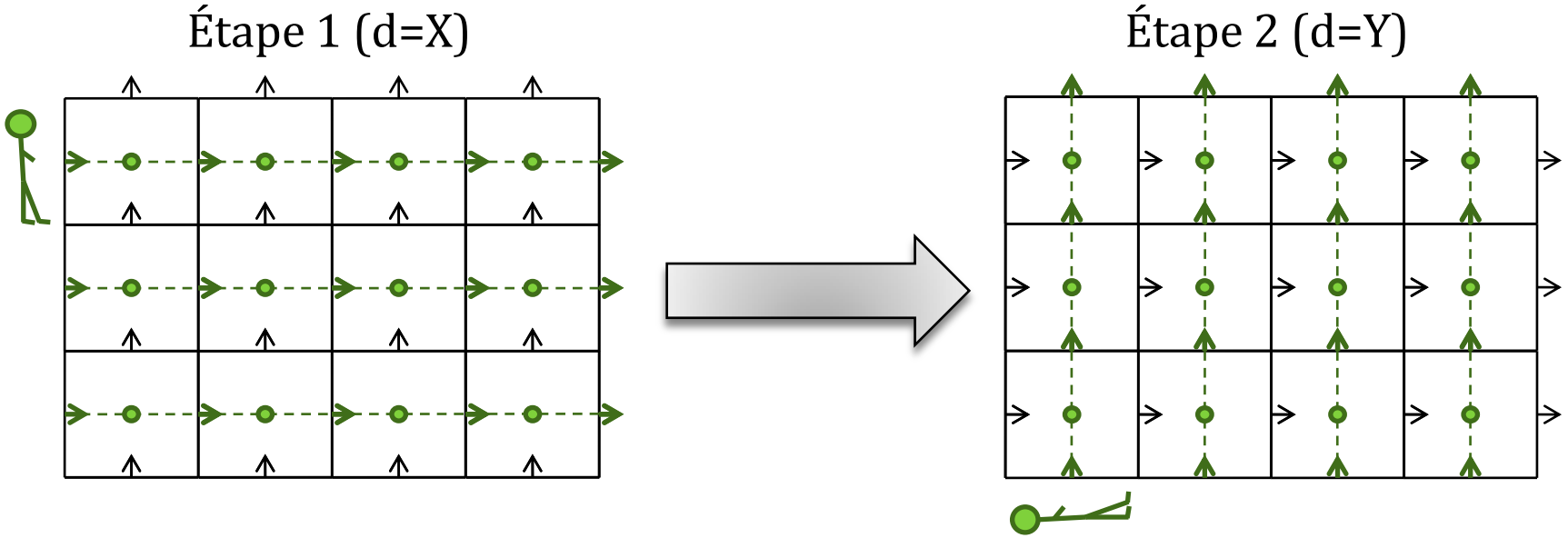
Librairies

- cuBLAS
- cuFFT

4. Parallélisation d'un code industriel d'algèbre linéaire

1. Introduction
2. DSL de l'algèbre linéaire creux structuré : Legolas++
3. Programmation de processeurs parallèles
4. **Parallélisation d'un code industriel d'algèbre linéaire**
   - Principes
   - Code actuel avec Legolas++
   - Code parallélisé avec TBB
   - Code parallélisé avec CUDA
   - Résultats
5. Conclusion

4. Parallélisation d'un code industriel d'algèbre linéaire : principes



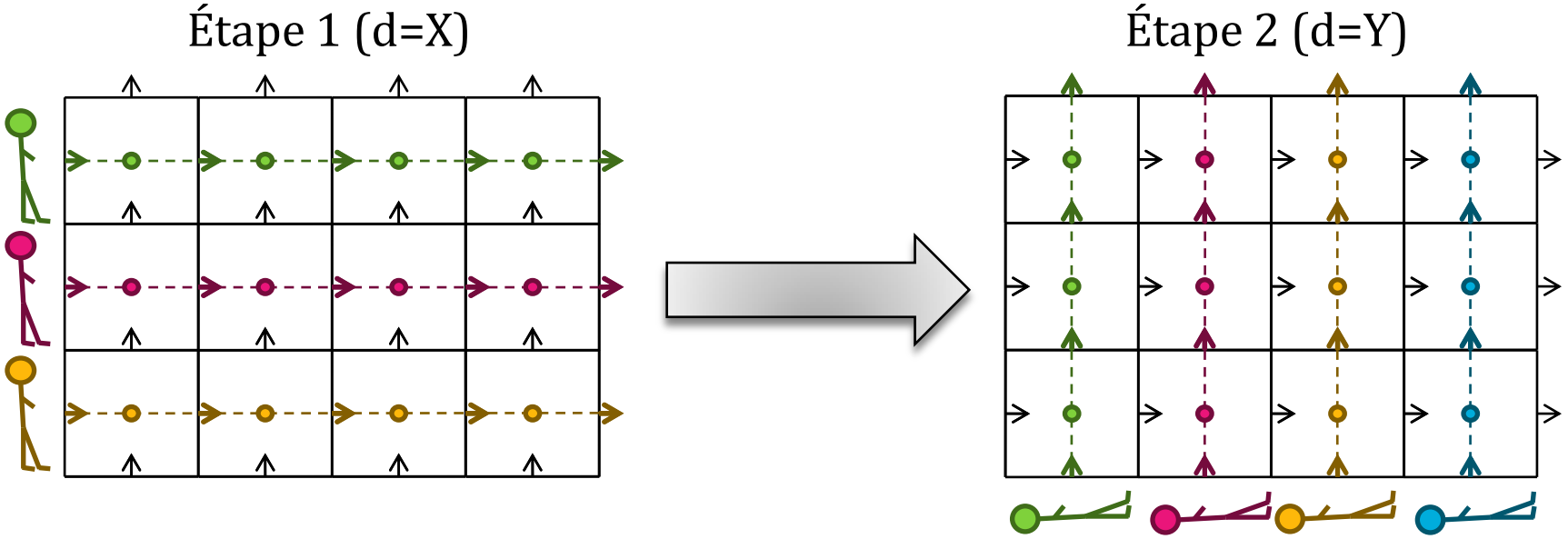
À chaque étape, on effectue des opérations vectorielles ou matricielles :

- f1(dir, vect)
- f2(dir, vect)
- f3(dir, vect)
- f4(dir, vect)

4. Parallélisation d'un code industriel d'algèbre linéaire : code Legolas++

```
for ( int iter = 0 ; iter < iteractionMax ; iter++)
    for ( int dir = 0 ; dir < 3 ; d--) {
        f1(dir, vectComplet);
        f2(dir, vectComplet);
        f3(dir, vectComplet);
        f4(dir, vectComplet);
        reordonnancement(dir);
    }
```

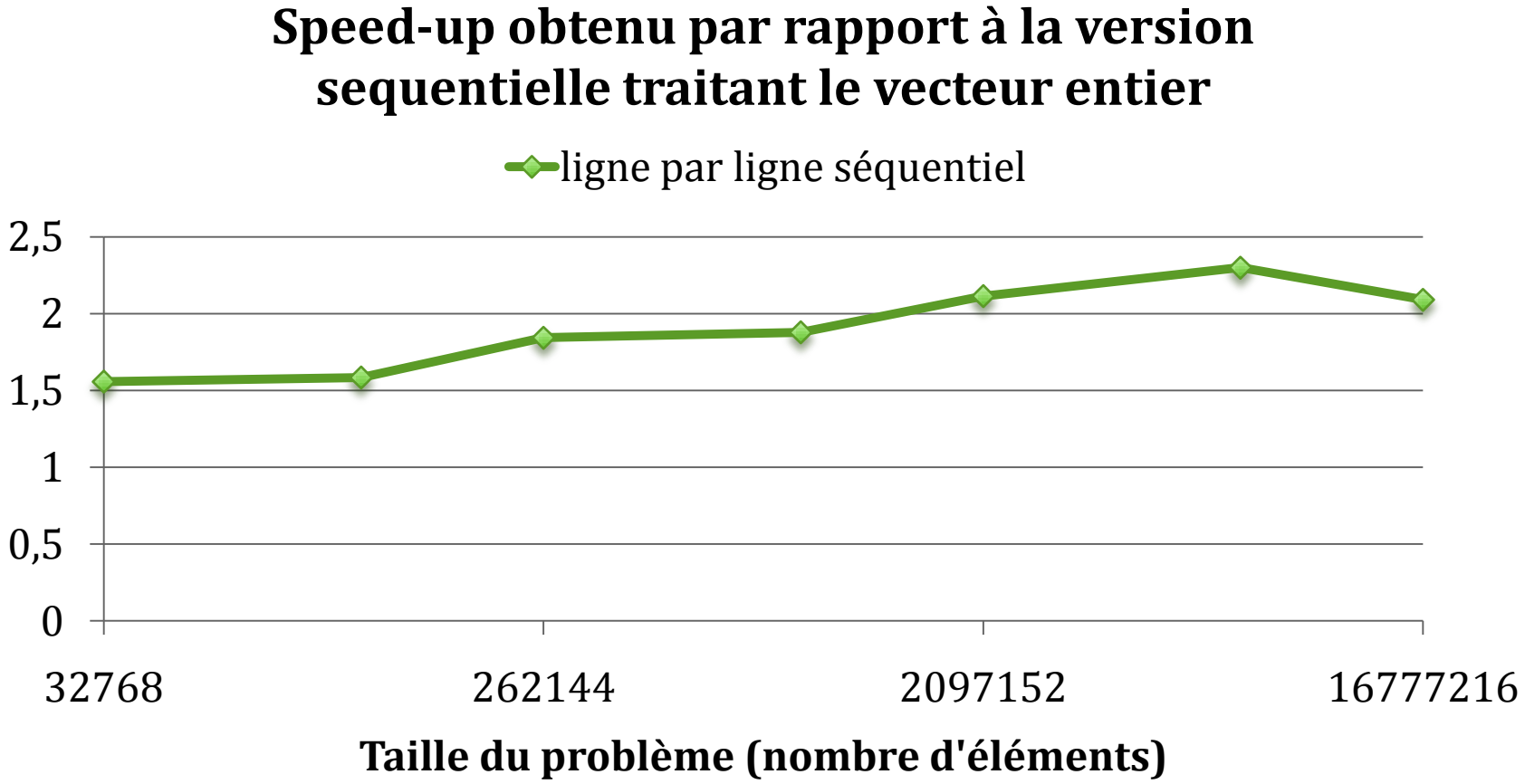
4. Parallélisation d'un code industriel d'algèbre linéaire : principes



4. Parallélisation d'un code industriel d'algèbre linéaire : code Legolas++

```
for ( int iter = 0 ; iter < iteractionMax ; iter++)
    for ( int dir = 0 ; dir < 3 ; d--) {
        for (int ligneIdx = 0 ; ligneIdx < nbLignes ; ligneIdx++){
            f1(dir, lignes[ligneIdx ]);
            f2(dir, lignes[ligneIdx ]);
            f3(dir, lignes[ligneIdx ]);
            f4(dir, lignes[ligneIdx ]);
        }
        reordonnancement(dir);
    }
```

4. Parallélisation d'un code industriel d'algèbre linéaire

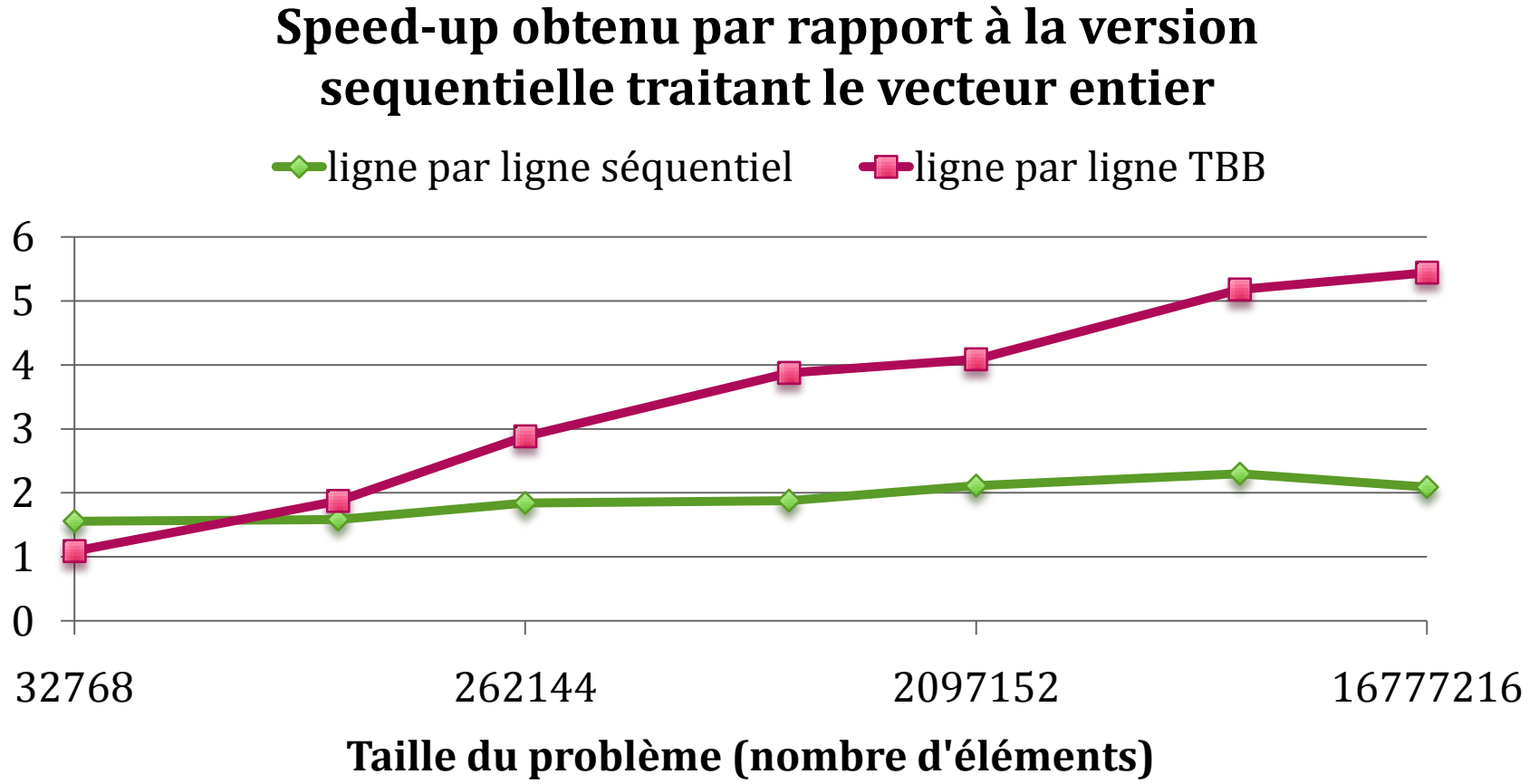


4. Parallélisation d'un code industriel d'algèbre linéaire : code Legolas++

```
for ( int iter = 0 ; iter < iteractionMax ; iter++)
    for ( int dir = 0 ; dir < 3 ; d--) {
        LigneParLigneFunctor functor(dir, lignes);
        const Range range(0, nbLignes );
        tbb : :parallel_for<Range,
            LigneParLigneFunctor>(range, functor);

        reordonnancement(dir);
    }
```

4. Parallélisation d'un code industriel d'algèbre linéaire



4. Parallélisation d'un code industriel d'algèbre linéaire

Pas de cache sur GPU !

➔Il faut optimiser le code à la main…

4. Parallélisation d'un code industriel d'algèbre linéaire

```
for (int i = 0 ; i < size ; i++)
    a[i] += b[i];
for (int i = 0 ; i < size ; i++)
    d[i] *= a[i];
```

6*size accès mémoire

```
for (int i = 0 ; i < size ; i++) {
    float a_i = a[i] + b[i];
    a[i] = a_i;
    d[i] *= a_i;
}
```

5*size accès mémoire

4. Parallélisation d'un code industriel d'algèbre linéaire

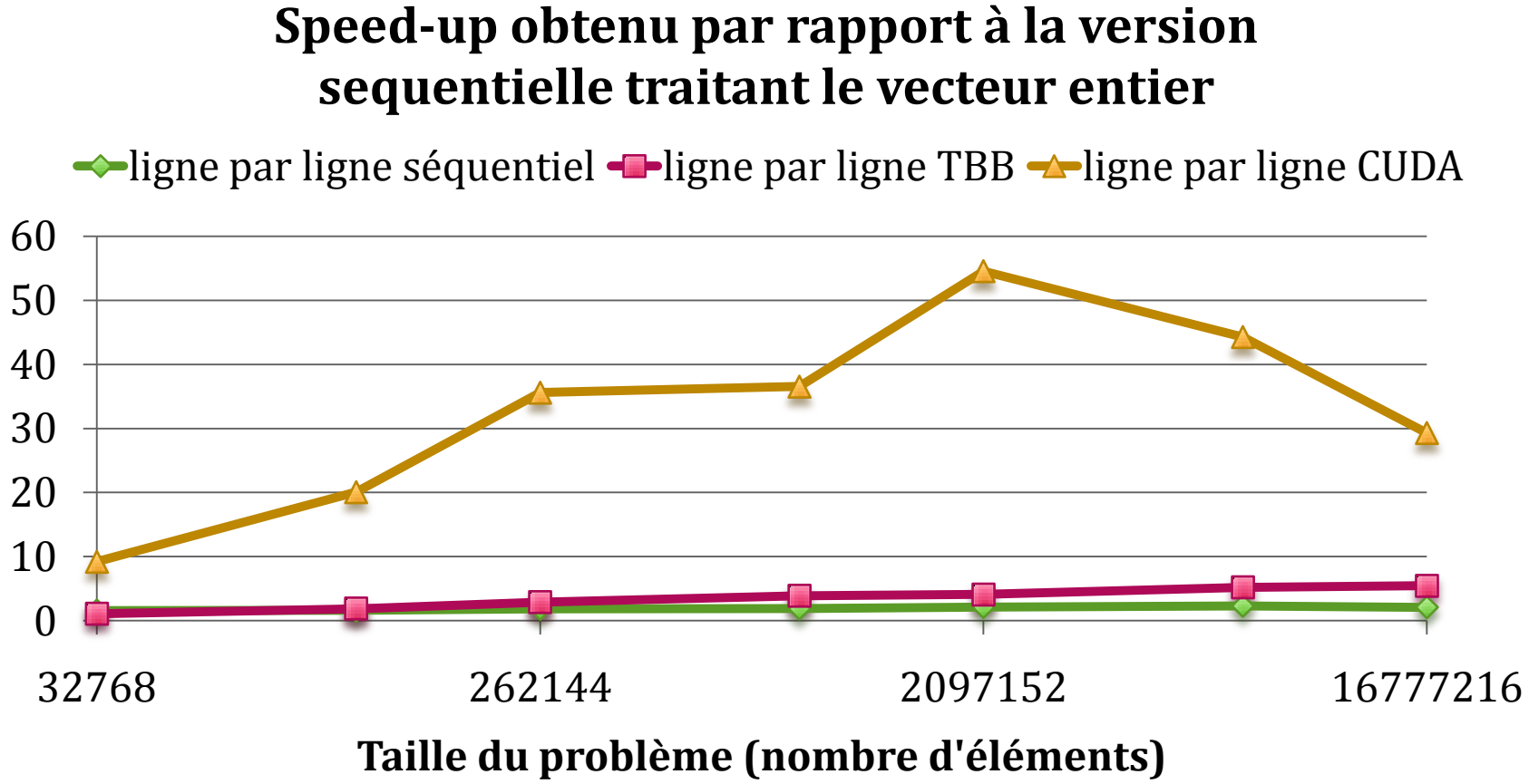
```
for (int i = 0 ; i < size ; i++)
    a[i] = b[i]-b[i+1];
```

3*size accès mémoire

```
float b_i = b[0];
for (int i = 0 ; i < size ; i++){
    float b_ip1 = b[i+1];
    a[i] = b_i - b_ip1 ;
    b_i = b_ip1;
}
```

2*size accès mémoire

4. Parallélisation d'un code industriel d'algèbre linéaire



5. Conclusion

- Introduction
- DSL de l'algèbre linéaire creux structuré : Legolas++
- Programmation de processeurs parallèles
- Parallélisation d'un code industriel d'algèbre linéaire
- **Conclusion**

Génération automatique de code multi-cible

- **NVCC : le compilateur CUDA**
- Modèle d'intégration de CUDA dans un code C++
- Solution envisagées

Génération de code multi-cible : NVCC le compilateur CUDA

Il n'y a pas de pile d'appels sur GPU. La programmation sur GPU est donc moins libre que sur CPU :

- Pas de fonctions récursives
- Pas de pointeurs de fonctions
- N'est pas compatible C++ pour le code des noyaux
- Supporte les templates avec certaines contraintes :
  - Pas de mot clé **typename**
- Le mot clé **typedef** n'est pas toujours géré correctement

Génération de code multi-cible : NVCC le compilateur CUDA

Il n'y a pas de pile d'appels sur GPU. La programmation sur GPU est donc moins libre que sur CPU :

- Pas de fonctions récursives

```
cuda_parallel_for< FunctorType< float > >
```

```
typedef FunctorType< float > Functor
cuda_parallel_for< Functor >
```

```
Typedef typename FUNCTOR::RealType;
```

Génération de code multi-cible : NVCC le compilateur CUDA

Il n'y a pas de pile d'appels sur GPU. La programmation sur GPU est donc moins libre que sur CPU :

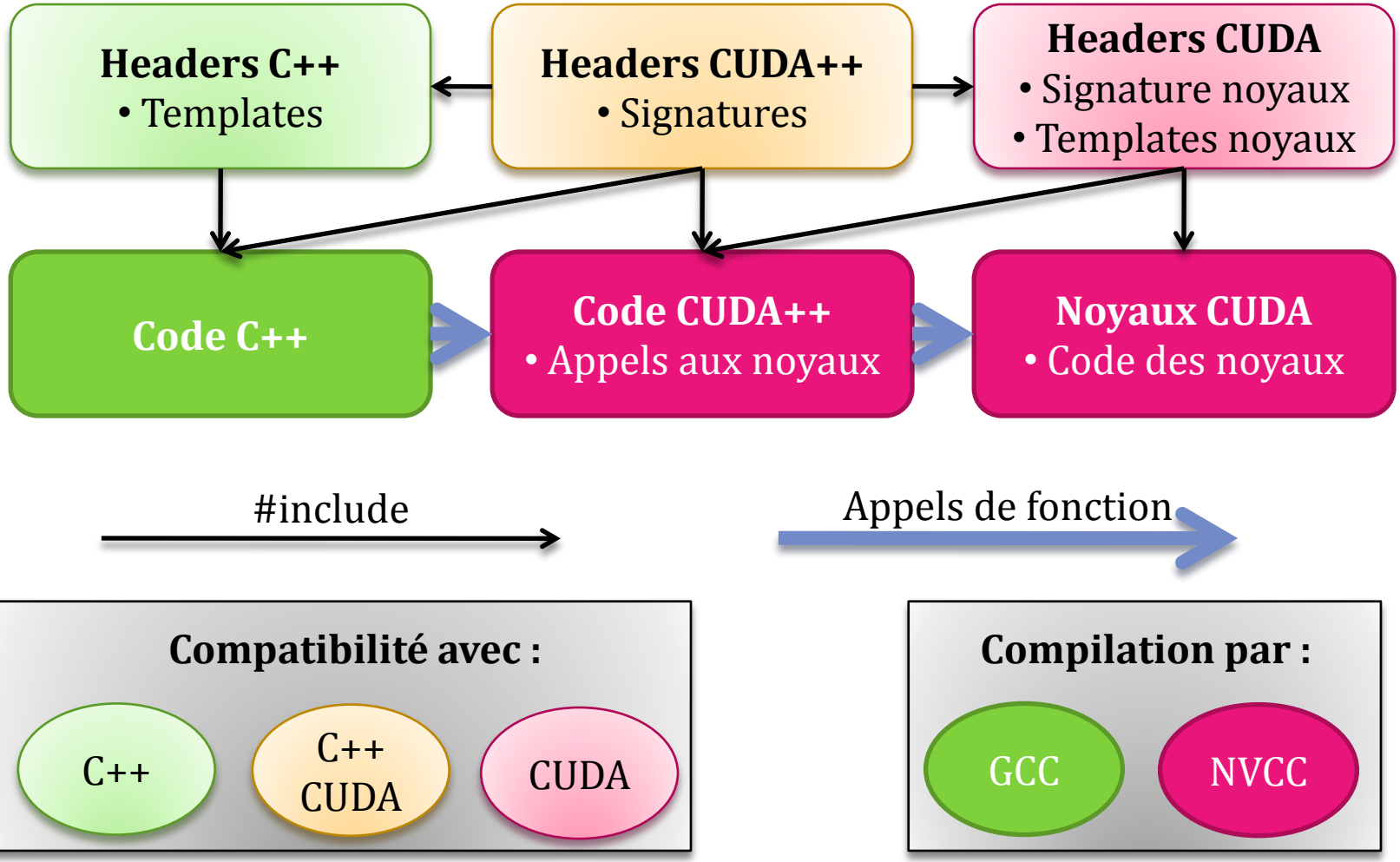
- Pas de fonctions récursives
- Pas de pointeurs de fonctions
- N'est pas compatible C++ pour le code des noyaux
- Supporte les templates avec certaines contraintes :
  - Pas de mot clé **typename**
- Le mot clé **typedef** n'est pas toujours géré correctement

➔ **Il n'est pas raisonnable de vouloir compiler une application C++ avec NVCC**

Génération automatique de code multi-cible

- NVCC : le compilateur CUDA
- **Modèle d'intégration de CUDA dans un code C++**
- Solution envisagées

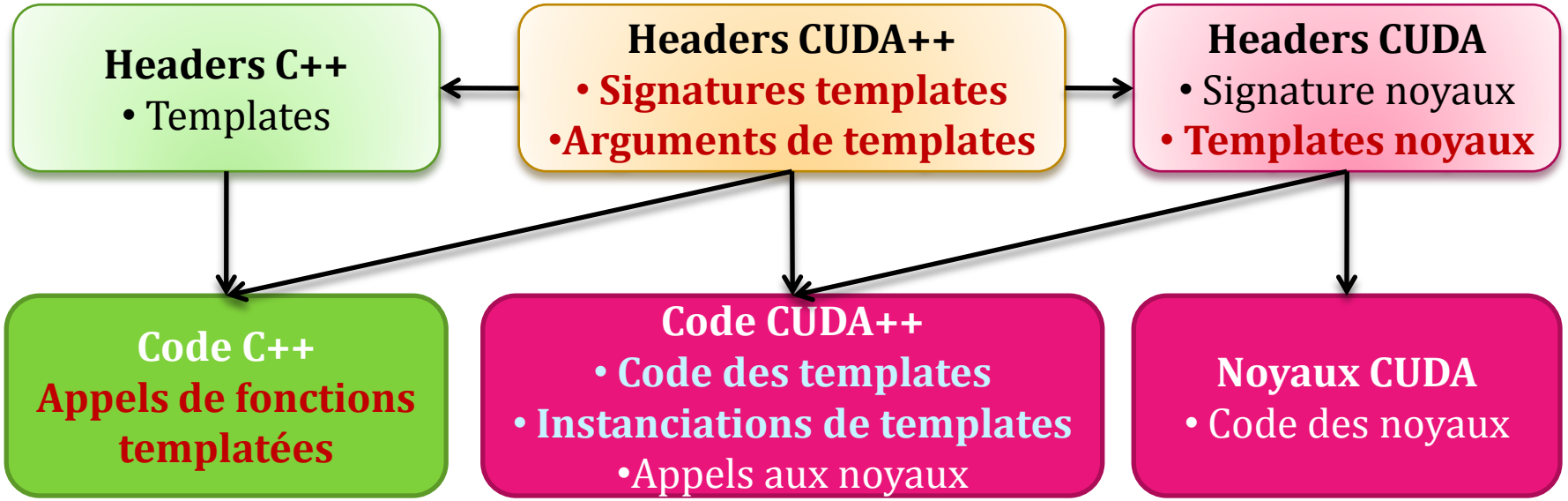
Génération de code multi-cible : modèle d'intégration CUDA/C++



Les accélérateurs de calcul

- NVCC : le compilateur CUDA
- Modèle d'intégration de CUDA dans un code C++
- **Solution envisagées**

Génération de code multi-cible : Solution envisagées



Génération de code multi-cible : Solution envisagées

Headers CUDA++

```
Template < typename FUNCTOR >
void cuda_parallel_for(const FUNCTOR & functor, const Range & range);

struct AxpyFunctor{
	float * x_;
	float * y_;
	void operator(const Range & range) const {
		for (int i = range.begin() ; i<range.end() ; i++)
			y[i]+=x[i];
	}
};
```

Génération de code multi-cible : Solution envisagées

Code C++

```
Int main ()
{
        ....
         AxpyFunctor functor;
        functor.x_ = ###;
        functor.y_ = ###;
        Range range(0, ###);
         cuda_parallel_for< AxpyFunctor >(functor, range);
        ...
};
```

Génération de code multi-cible : Solution envisagées

Code CUDA++

```
Template < typename FUNCTOR>
void cuda_parallel_for(const FUNCTOR & functor, const Range & range)
{
        …
        cuda_parallel_forKernel< Functor ><<< Dg, Db >>>(functor, range);
        …
};
```

Génération de code multi-cible : Solution envisagées

Headers CUDA++

```
Template < typename FUNCTOR>
void cuda_parallel_forKernel(const FUNCTOR functor, const Range range)
{
        …
        functor( localRange );
        …
};
```

Génération de code multi-cible : Solution envisagées

Code CUDA++

```
Template < typename FUNCTOR>
cuda_parallel_for(const FUNCTOR & functor, const Range & range)
{
        …
        cuda_parallel_forKernel< Functor ><<< Dg, Db >>>(functor, range);
        …
};

void instantiateTemplates()
{
        AxpyFunctor functor;
        Range range(0,0);
        cuda_parallel_for< AxpyFunctor >(functor, range);
};
```