Triton : le langage dédié d'OpenAI pour le deep learning

L'heure de la maturité pour Triton ? Le projet, que porte OpenAI, vient de passer en version 1.0. Sa principale brique : un langage dédié ciblant le *deep learning*. Objectif : faciliter l'implémentation d'opérations non optimisées pour les bibliothèques de référence.

Triton arrive sur un terrain déjà bien occupé. Deux approches majeures s'y sont développées. D'un côté, la compilation polyédrique, adoptée notamment par Tiramisu et Tensor Comprehensions. Elle représente les programmes de sorte que le flux de contrôle est prédictible. Cela favorise les transformations lors de la compilation, en intégrant le parallélisme et la localité des données.
La méthode permet par ailleurs de contrôler la préservation sémantique. Elle présente en revanche une consommation importante de ressources – doublée de procédures coûteuses d'*autotuning* – et une certaine inertie face au codage parcimonieux.

L'autre approche, adoptée entre autres par Halide et TVM, implique les langages de planification. Ces derniers mettent en œuvre le principe de séparation des préoccupations : ils dissocient l'implémentation de la conception. On n'a donc à écrire qu'une seule fois un algorithme, puis on l'optimise à part. Avec la possibilité de spécifier des éléments qu'un compilateur polyédrique n'aurait pas pu déterminer à partir de l'analyse statique des flux de données.
Les systèmes fondés sur ce paradigme présentent eux aussi des limites face au codage parcimonieux. Il peut en résulter des contraintes importantes dans la planification.

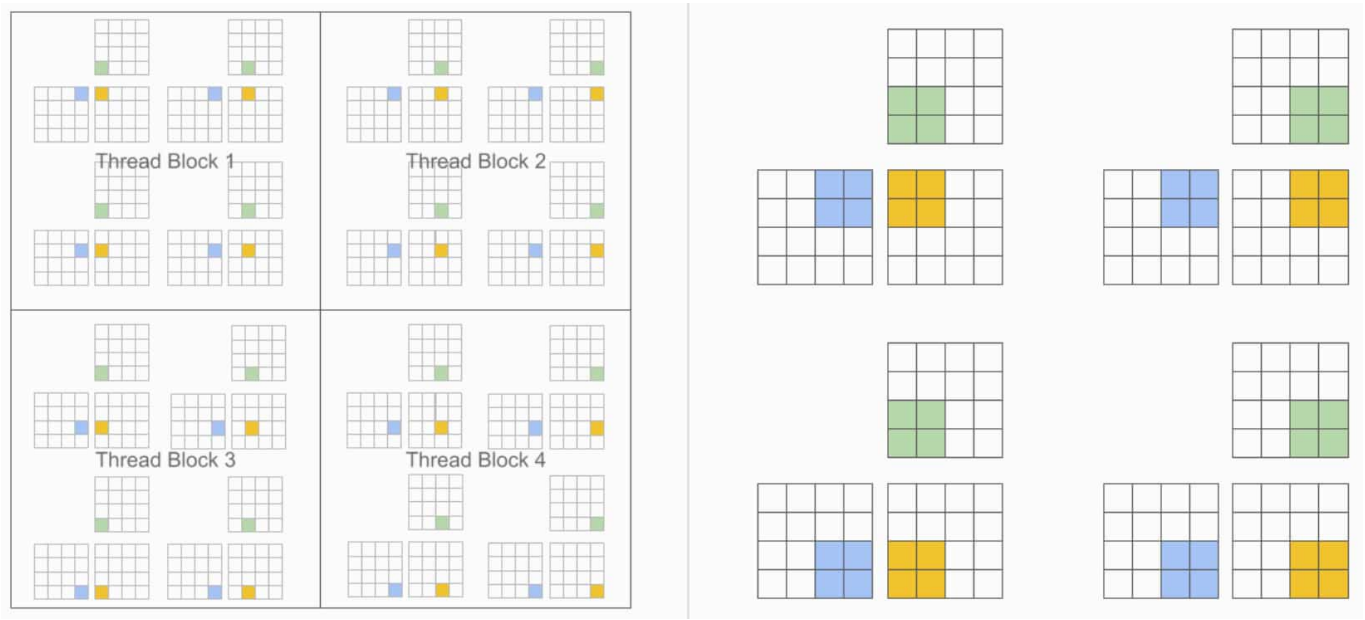
Triton vs CUDA : une histoire de blocs

Triton repose sur une variante du modèle d'exécution SMPD (Single Program, Multiple Data). Avec elle, ce sont pas les tâches qui sont architecturées en blocs, mais les programmes. Soit l'inverse de CUDA (à gauche sur l'exemple ci-dessous, une multiplication matricielle).

```
#pragma parallel
for(int m = 0; i < M; m++)
#pragma parallel
for(int n = 0; j < N; n++){
  float acc = 0;
  for(int k = 0; k < K;k ++)
    acc += A[i, k]* B[k, j];

  C[i, j] = acc;
}
```

```
#pragma parallel
for(int m = 0; m < M; m += MB)
#pragma parallel
for(int n = 0; n < N; n += NB){
  float acc[MB, NB] = 0;
  for(int k = 0; k < K; k += KB)
    acc +=  A[m:m+MB, k:k+KB]
          @ B[k:k+KB, n:n+NB];
  C[m:m+MB, n:n+NB] = acc;
}
```



Cette approche se révèle particulièrement flexible lorsqu'on a à traiter des matrices creuses (qui comportent beaucoup de coefficients nuls). En analysant les flux de données au niveau des blocs, le compilateur Triton est capable d'automatiser un certain nombre d'opérations importantes. Dont la coalescence (agrégation des transferts DRAM), la vectorisation et la planification des cœurs.

Triton fonctionne pour le moment sur des GPU NVIDIA. Des travaux sont en cours pour l'adapter aux GPU AMD et aux CPU. On évitera la confusion avec un autre projet du même nom, made in NVIDIA. Il s'agit d'un moteur d'inférence. La plate-forme de cybersécurité Morpheus, actuellement en accès anticipé, en fait usage.

Illustration principale © kras99 – Adobe Stock