Verrou : Instrumentation sur GPU

Estelle OLIVEIRA
Formation : CHPS – Université Versailles Saint-Quentin

Encadrant : Bruno LATHUILIERE

Verrou : Instrumentation sur GPU

Sommaire

1.

Contexte - Verrou

Qu’est ce que l’erreur flottante ?

Réels : espace infini, continu
Flottants : espace fini, discret
→ représentation binaire limitée
→ approximation (erreur d’arrondi)

Floating point computation ≠ Real computation

$a \oplus b \neq a + b$
$(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$

1 erreur sur **1** opération peut être négligeable **MAIS**

Accumulation d’erreur

Exemple

$\sum_{k=0}^{10^8} 0.1 = 10^7$ *Précision infinie*

$\sum_{k=0}^{10^8} 0.1 = 2.097 * 10^6$ *Précision flottante 32bits*

Cancellation

Exemple

$(1. - 10^{-8}) - 1. = -10^{-8}$ *Précision infinie*

$(1. - 10^{-8}) - 1. = 0.$ *Précision flottante 32bits*

→ important de connaître l’écart entre le résultat théorique en précision infinie et le résultat en précision limitée

→ **vérification des OCS**

Verrou

Objectif : Estimation de l'erreur flottante et localisation

→ Instrumentation **binaire** via le **framework** valgrind

- Support de tous les languages de programmation C, C++, Fortran …

- Pas de recompilation de code

- Facilité d'utilisation

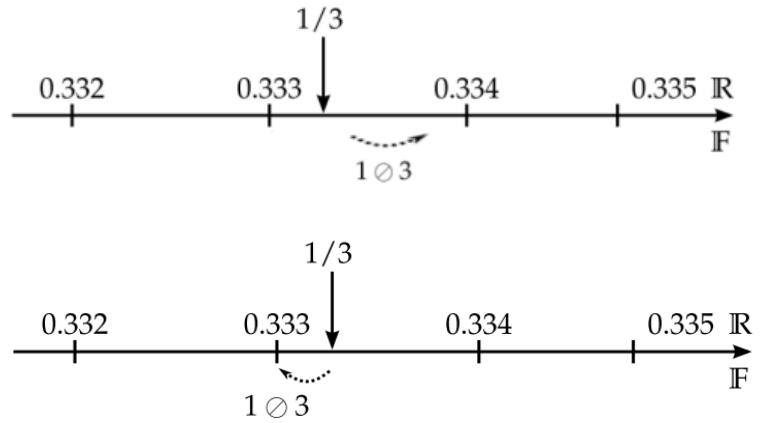
```
valgrind –tool=verrou --rounding-mode=random ./mycode
```

→ Arithmétique stochastique

Modes d’arrondis

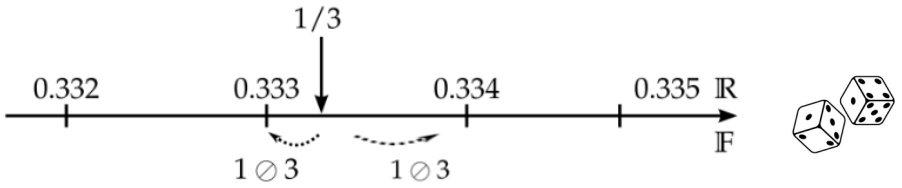
Arrondis déterministes :

- Nearest
- Upward
- Downward
- Round-to-zero



Arrondis stochastiques :

- Random
- ...



Accélérateur GPU

De plus en plus de codes sont portés sur carte graphique GPU
- Performance
- Consommation énergétique

→ programmation sur GPU

	Nvidia	AMD
Langage bas niveau	**Cuda**	Hip
Outils portables	**Kokkos** OpenMP target	

Points critiques :

- Différence d'**algorithmes** par rapport aux CPU
- Différence de **résultats** par rapport aux CPU

→ Besoin d'estimer l'erreur flottante sur architecture GPU

2.

Instrumentation - GPU

Objectifs

Dans un premier temps on veut recréer toutes les fonctions principales de verrou pour avoir tous les outils nécessaires au développement.

- Remplacer les opérations flottantes par nos implémentations
  - Implémenter modes d'arrondis déterministes
  - Implémenter modes d'arrondis stochastiques
    - Implémenter générateur aléatoire sur GPU (random ...)
    - Implémenter fonctions de hashage (random-det ...)
- Récupérer les numéros de lignes des opérations pour le besoin de localisation

Utilisation de nvbit

Outil d'instrumentation de nvidia

→ C'est le seul outil disponible

→ On bénéficie du retour d'expérience de GPU-FPX, exceptions arithmétique flottante (NaN, DIV0...)

Nvbit est un projet qui se base sur le **SASS**, assembleur nvidia de plus bas niveau

→ lecture / écriture dans les registres

→ lecture SASS et injection de fonction par instruction

→ suppression d'instruction

Lancement de verrouGPU

```
LD_PRELOAD=/path/to/verrouGPU/verrouGPU.so ROUNDING=nearest    ./mycode
```

SASS nvidia

Précision x Opération x PARAM2 x PARAM3(si présent) x signe x mode arrondi

→ environ 800 fonctions

→ génération de code et templates

Instrumentation

→ **Boucle kernels / fonctions**
→ **Boucle instructions**

- Code-opération
  Précision x Opération

- Injection de fonction associée
TypeP Operation_sign1_sign2_rounding

- Retirer l'instruction initiale

Cas d'une addition flottante, avec 2 registres

```
for (auto f : related_functions) {
    for (auto i : instrs) {

        if (i->getIdx() >= instr_begin_interval && i->getIdx() < instr_end_interval) {
            std::string opcode = i->getOpcode();

            if (opcode == "FADD" || opcode == "FADD.FTZ") {
            const InstrType::operand_t *op0 = i->getOperand(0);
            const InstrType::operand_t *op1 = i->getOperand(1);
            const InstrType::operand_t *op2 = i->getOperand(2);

                if ((op2->type  == InstrType::OperandType::REG)&&(op1->type == InstrType::OperandType::REG)) {
                    nvbit_insert_call(i, "Wrap_RegAdd_Plus_Plus_nearest", IPOINT_AFTER);
                    nvbit_add_call_arg_guard_pred_val(i);
                    nvbit_add_call_arg_const_val32(i, op1->u.reg.num);
                    nvbit_add_call_arg_const_val32(i, op2->u.reg.num);
                    nvbit_add_call_arg_const_val32(i, op0->u.reg.num);
                    nvbit_remove_orig(i);
                }
            }
        }
    }
}
```

code simplifié

3.

Fonctions injectés - GPU

Fonctions injectées – via nvbit (nvidia instrumentation tool)

FADD float c = float a + float b

1. Lecture de registre → paramètres

    1.1 Registres, Registres Uniformes

    1.2 Conversions

2. Opération que nous voulons faire :

    __fadd_rn(a, b)

3. Résultat réécrit dans le registre de destination

```
18  template<Sign register_sign1, Sign register_sign2>
19  __device__ void _wrap_reg(int predicate, int reg_1_num, int reg_2_num, int reg_dst_num) {
20      if (!predicate) return;
21      uint32_t reg1_val_int = nvbit_read_reg(reg_1_num);
22      float reg1_val_float = __int_as_float(reg1_val_int);
23      reg1_val_float = affectsign<register_sign1>(reg1_val_float);
24      uint32_t reg2_val_int = nvbit_read_reg(reg_2_num);
25      float reg2_val_float = __int_as_float(reg2_val_int);
26      reg2_val_float = affectsign<register_sign2>(reg2_val_float);
27
28
29      float result_float = __fadd_rn(reg1_val_float, reg2_val_float);
30
31      uint32_t result_int = __float_as_int(result_float);
32      nvbit_write_reg(reg_dst_num, result_int);
33  }
```

Fonction simplifiée

__fadd_rn(a, b) : nearest
__fadd_ru(a, b) : upward
__fadd_rd(a, b) : downward
__fadd_rz(a, b) : round-to-zero

4.

Résultats

Résultats

Tests Unitaires Cuda :

- opérations principales
- sans notion de performances

Tests Kokkos : *AXPY*

Mesure	Temps (s)	ratio
CPU natif	80.41	1.00
Valgrind –tool=none	348.25	4.33
Verrou, nearest	1028.75	12.79
Verrou, upward	1386.92	17.25
GPU natif	22.65	1.00
VerrouGPU, nearest	393.04	17.35
VerrouGPU, upward	391.75	17.29

→ Mode d'arrondi demande plus de temps d'instrumentation

→ Plus d'influence du mode d'arrondi sur le temps d'exécution

VerrouGPU upward est plus rapide que Verrou upward

Retour d’expérience sur nvbit

Difficultés rencontrées

- Récupération de lignes (localisation)
    - incompatibilité de version entre les outils : nvcc, cuda, nvbit, gcc
    - pas de documentation nvbit, pas de sources

- Générateur aléatoire (modes d’arrondis stochastiques)
    - incompatibilité des options de compilation de l’outil et de la librairie de générateur aléatoire fournit par Cuda

- Compatibilité nvbit et valgrind

- SASS bas niveau
    - division
    - parallel reduce Kokkos

Les bonnes surprises

- “*Facilité*” d’instrumentation du SASS
- Bonne performance

5.

Conclusion

Conclusion

→ **Objectifs atteints :**

- Remplacer les opérations par nos implémentations avec des performances acceptables

- Implémenter les modes d'arrondis déterministes : nearest, upward, downward, round-to-zero

→ **Perspectives :**

- Étendre base de cas tests

- Trouver des solutions aux difficultés rencontrées

- Appliquer sur code industriel

Merci

Contacts

OLIVEIRA Estelle
estelle.oliveira@edf.fr

Annexe I – Fonctions injectées (float)

1. Lecture de registre → paramètres
   1.1 Registres, Registres Uniformes
   1.2 Conversions
2. Opération que nous voulons faire :
   __[prec][op]_[rounding](a, b, c)
3. Résultat réécrit dans le registre de destination

```
template<Op operation, Rounding rounding_mode, typename prec, Sign register_sign1, Sign register_sign2>
__device__ void _wrap_reg(int predicate, int reg_1_num, int reg_2_num, int reg_dst_num) {
    if (!predicate) return;
    uint32_t reg1_val_int = nvbit_read_reg(reg_1_num);
    float reg1_val_float = __int_as_float(reg1_val_int);
    reg1_val_float = affectsign<register_sign1>(reg1_val_float);
    uint32_t reg2_val_int = nvbit_read_reg(reg_2_num);
    float reg2_val_float = __int_as_float(reg2_val_int);
    reg2_val_float = affectsign<register_sign2>(reg2_val_float);



    float result_float = apply<operation, rounding_mode, prec>(reg1_val_float, reg2_val_float);   //proceed the add

    uint32_t result_int = __float_as_int(result_float);
    nvbit_write_reg(reg_dst_num, result_int);
}
```

Annexe II – Fonctions injectées (double)

1. Lecture de registre → paramètres
   1.1 Registres, Registres Uniformes
   1.2 Conversions
2. Opération que nous voulons faire :
   __[prec][op]_[rounding](a, b, c)
3. Résultat réécrit dans le registre de destination

```
20  template <Op operation, Rounding rounding_mode, typename prec, Sign register_sign1, Sign register_sign2>
21  __device__ void _wrap_doublereg(int predicate, int reg_1_num, int reg_2_num, int reg_dst_num) {
22      if (!predicate) return;
23      uint32_t reg1_lowval_int = nvbit_read_reg(reg_1_num);
24      uint32_t reg1_highval_int = nvbit_read_reg(reg_1_num+1);
25      uint64_t reg1_val_int = ((uint64_t)reg1_highval_int << 32) | reg1_lowval_int;
26      double reg1_val_dbl = *(double*)&reg1_val_int;
27      reg1_val_dbl = affectsign<register_sign1>(reg1_val_dbl);
28
29      uint32_t reg2_lowval_int = nvbit_read_reg(reg_2_num);
30      uint32_t reg2_highval_int = nvbit_read_reg(reg_2_num+1);
31      uint64_t reg2_val_int = ((uint64_t)reg2_highval_int << 32) | reg2_lowval_int;
32      double reg2_val_dbl = *(double*)&reg2_val_int;
33      reg2_val_dbl = affectsign<register_sign2>(reg2_val_dbl);
34
35      double result_dbl = apply<operation, rounding_mode, prec>(reg1_val_dbl, reg2_val_dbl);
36      uint64_t result_int64 = *(uint64_t*)&result_dbl;
37
38      uint32_t result_intlow = (uint32_t)(result_int64 & 0xFFFFFFFF);
39      uint32_t result_inthigh = (uint32_t)(result_int64 >> 32);
40
41      nvbit_write_reg(reg_dst_num, result_intlow);
42      nvbit_write_reg(reg_dst_num+1, result_inthigh);
43  }
```