Corrigé et commentaires

Exercice 1. Étude comparative de la performance arithmétique des routines BLAS

L'objectif de cet exercice était d'évaluer et de comparer la performance des plusieurs routines BLAS : `cblas_dgemm`, `cblas_sgemm`, `cblas_dgemv` et `cblas_sgemv`. Les deux premières effectuent une opération gemm ($\alpha\mathbf{AB}+\beta\mathbf{C}$), tandis que les deux dernières effectuent une opération gemv ($\alpha\mathbf{Ax}+\beta\mathbf{y}$). Ces routines effectuent $2n^3+2n^2$ et $2n^2+2n$ opérations à virgule flottante (*i.e* FLOP), respectivement.

La performance peut s'évaluer de différentes façons : temps de calcul *(en secondes)*, complexité *(en nombre d'opérations)* ou débit arithmétique *(en opération par seconde)*.

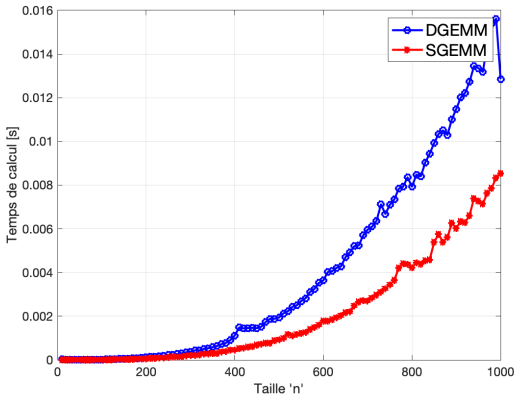
Modification du code

On reprend le code `dgemmMKL.c`. Dans ce code, j'ai ajouté une boucle pour itérer sur les valeurs de n *(pour automatiser l'étude)*. Pour chaque valeur de n, la fonction `cblas_dgemm` est appelée plusieurs fois pour ensuite évaluer un temps de calcul moyen *(les temps de calcul étant très petit, il est nécessaire de répéter l'opération plusieurs fois pour avoir un temps fiable)*. J'ai également implémenté le calcul du nombre d'opérations et du debit arithmétique. Même chose pour les fonctions `cblas_sgemm`, `cblas_dgemv` et `cblas_sgemv`.

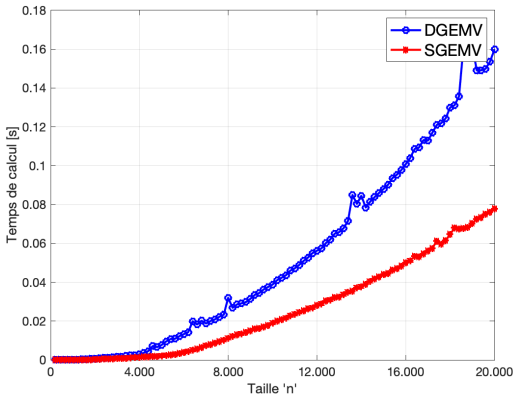
Étude du temps de calcul

Voici les temps de calcul pour les différentes routines en fonction de la taille n des matrices ($n \times n$). Ces temps on été obtenu avec mon ordinateur *(MacBook Pro, processeur Intel Core i7, quad-core, 2.8GHz)*.

GEMM : $\alpha\mathbf{AB}+\beta\mathbf{C}$



GEMV : $\alpha\mathbf{Ax}+\beta\mathbf{y}$

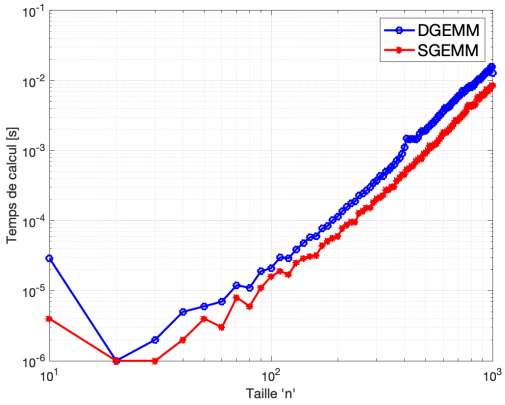


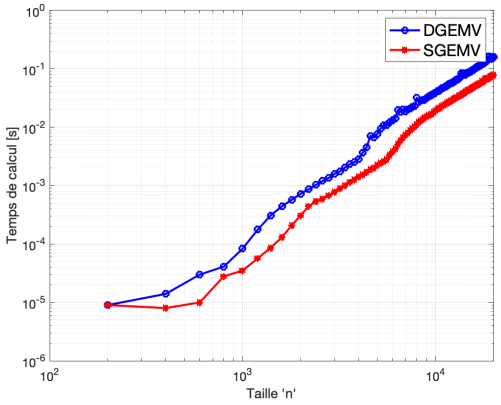
- Le temps de calcul augmente évidemment avec n. Les courbes fluctuent parfois. Ce n'est pas grave, c'est probablement du à d'autres tâches effectuées par mon ordinateur, qui ont probablement ralenti les calculs.
- Dans les deux cas (GEMM et GEMV), les opérations sont deux fois plus lentes en double précision qu'en simple précision. En terme de nombre de bits, il y a deux fois plus d'opération à réaliser en double précision qu'en simple précision.

Ces figures ne permettent pas d'en dire plus.

Voici les mêmes figures en échelle logarithmique.

GEMM : $\alpha\mathbf{AB} + \beta\mathbf{C}$ GEMV : $\alpha\mathbf{Ax} + \beta\mathbf{y}$





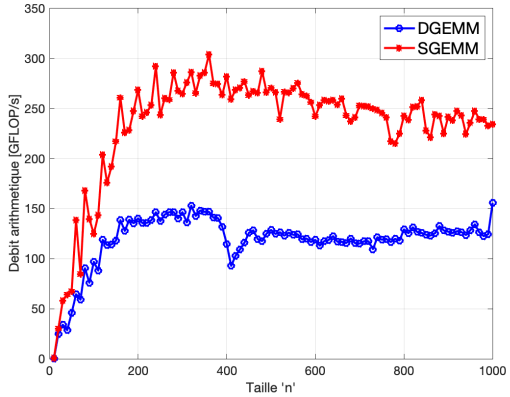
En analysant la partie droite des courbes, on observe que la pente des droite nous permet de retrouver la complexité des opérations (d'ordre 3 pour GEMM, et d'ordre 2 pour GEMV).

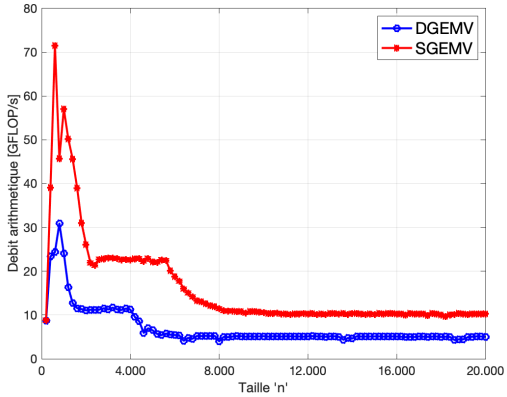
Il est important de noter que la complexité ne permet que de comparer des algorithmes entre eux de manière abstraite. Elle ne tient pas compte des aspects liés à l'implémentation ou des machines de calcul.

Étude du débit arithmétique

Le débit arithmétique est le rapport du nombre d'opération à virgule flottante *(FLOP, floating point operation)* par la durée nécessaire pour effectuer ces opération. Il s'exprime en *FLOP/s* ou en *GFLOP/s*, avec 1 $GFLOP/s = 10^9$ $GFLOP/s$.

GEMM : $\alpha\mathbf{AB} + \beta\mathbf{C}$ GEMV : $\alpha\mathbf{Ax} + \beta\mathbf{y}$





- On observe une phase 'étrange' au début des courbes, lorsque les matrices sont petites (pour $n < 200$ à gauche, et pour $n < 6000$ à droite). Le comportement n'est pas le même à gauche et à droite, et il est difficile à interpréter. Ce qui est certain, c'est que les matrices sont petites, et peuvent facilement tenir dans les caches du processeur. À l'inverse, il y a peu de calcul à faire, et les unités de calcul pourraient de pas être occupées suffisament. À ce niveau, c'est intéressant à remarquer, mais l'étude complète de ce phénomène dépasse le cadre du cours.
- Pour des grandes matrices (pour $n > 200$ à gauche, et pour $n > 6000$ à droite), les débits restent relativement stables. Le débit est deux fois plus rapide en simple précision qu'en double précision *(ce qui est cohérent avec notre observation sur le temps de calcul)*.

– Il est intéressant de remarquer que le débit est beaucoup plus important pour les routines dgemm (de l'ordre de 100-200 GFLOP/s) par rapport aux routines dgemv (de l'ordre de 5-10 GFLOP/s). Là aussi, c'est compliqué à expliquer. Le calcul correspondant aux opérations dgemm sont plus 'intenses' que pour les opérations dgemv. La complexité est de n^3 dans le premier cas et de n^2 dans le deuxième cas, alors que la quantité de donnée est de l'ordre de n^2 dans les deux cas. De nouveau, nous touchons à la limite du cadre de cours.
– Ce faut retenir : le débit arithmétique est une mesure de l'efficacité de votre implémentation. Il va dépendre du type d'algorithme et de la qualité de l'implémentation. Il est difficile d'avoir des implémentation efficace pour beaucoup d'algorithmes.

Remarque avancée

Les résultats ci-dessus utilisent le calcul parallèle à mémoire partagée. Même si vous n'utilisez pas l'option de compilation `-qopenmp`, la librairie mkl est compilée en parallèle, et utilise le nombre de threads par défaut *(4 dans mon cas)*.

Exercice 2. Code de différences finies – Analyse du rapport de compilation et vectorisation

Analyse du rapport de compilation

L'objectif de cet exercice était de lire le rapport de compilation du code `diffusionIcc.c` et de repérer des informations intéressantes. Pour le dernier niveau d'optimisation (`-O3`), le rapport est à la suite de ce document.

Quelques observations :

– Pratiquement aucune optimisation n'est réalisée avec `-O0`. Avec `-O1`, des `inline` sont réalisés. Avec `-O2`, la vectorisation est activée, mais uniquement la boucle d'initialisation est vectorisée. A priori, il n'y a pas de différence avec `-O3`. Nous allons rester sur le niveau `-O3` dans la suite.
– Ligne 22 *(Inline)* : Les appels à la fonction `switchPointer`, qui est très courte, ont été remplacés par les opérations que la fonction effectuait. C'est une petit optimisation qui permets de réduire des allers-retours pendant la lecture du programme.
– Ligne 39 *(Vectorization)* : Le rapport indique que la boucle est vectorisée. Les indications *peeled loop* et *remainder loop* signifient que les premières/dernières itérations ne seront pas vectorisées de la même façon que les autres. L'objectif est d'avoir des donnés/opérations bien alignés la ligne de cache avec les bonnes tailles de vecteurs. Pour plus de détails : lien
– Lignes 50-51-52 *(Unroll)* : De façon un peu étrange, les informations sur l'optimisation des boucles des fonctions `implementationX()` se retrouvent ici. En lisant le rapport, on note que des boucles ont été déroulées. Les boucles ne sont pas vectorisées, alors qu'elles pourraient l'être *(le compilateur n'a pas activé la vectorisation)*.

Nous n'irons pas plus en détail dans l'analyse de ce rapport. Cet exercice vous a permis, j'espère, de comprendre un peu plus la logique du compilateur. Lorsque vous aurez besoin de la confirmation qu'une optimisation a bien été prise en compte, vous pourrez utiliser cet outil.

Remarque : Si vous ajouter de la parallélisation à mémoire partagée, et l'option de compilation `-qopenmp`, vous aurez de nouvelles informations sur la parallélisation des opérations.

Vectorisation des boucles

La fonction correspondant à l'implémentation 3 est reprise ci-dessous :

```
void implementation3(int T, int N, double* C, double* Cnew){
  double dx = 1./(double)(N-1);
  double dt = 0.2*dx*dx;
  double coef1 = dt/(dx*dx);
  double coef2 = 1 - 4*coef1;

  for(int n=0; n<T; n++){
    for(int i=1; i<(N-1); i++)
      for(int j=1; j<(N-1); j++)
        Cnew[N*i+j]
          = coef2 * C[N*i+j]
          + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j] + C[N*i+(j+1)] + C[N*i+(j-1)] );
    switchPointer(&C,&Cnew);
  }
}
```

Les deux boucles intérieures *(en i et j)* correspondent à des opérations parallélisables *(i.e. les opérations des différentes itérations peuvent se faire 'en concurrence' : il n'y a pas de dépendance entre elles)*, par contre la boucle extérieure *(en n)* ne peut pas être parallélisée *(il s'agit d'une boucle 'temporelle', où chaque itération dépend de la précédente)*.

Les boucles intérieures ne sont pas vectorisées car le compilateur ne sait pas si les données correspondant aux pointeurs `C` et `Cnew` sont indépendantes ou non. Pour indiquer au compilateur qu'une boucle peut être vectorisée, en lui indiquant que les données sont indépendantes, on utilise `#pragma ivdep`. *(Voir slide 30 du cours 2.)*

Dans le code `diffusionIccVec.c`, j'ai testé l'ajout du `#pragma ivdep` devant chacune des trois boucles. Voici le résultat :

- Devant la boucle n : la boucle n'est pas vectorisée *(mécanisme de protection du compilateur ?)*, et temps de calcul de 1.6 secondes.
- Devant la boucle i : la boucle est vectorisée et temps de calcul de 1.9 secondes.
- Devant la boucle j : la boucle est vectorisée et temps de calcul de 1.4 secondes.

La vectorisation (ou non-vectorisation) est confirmée par le rapport de compilation.

Il faut donc suivre la règle de bonne pratique : vectoriser la boucle la plus intérieure.

Exercice 3. Code dgemm – Optimisation et analyse du rapport de compilation

Pas de retour sur cet exercice.

```
1       // Compilation:
2       //   icc -O0 -std=c99 -mkl diffusionIcc.c
3       //   icc -O1 -std=c99 -mkl diffusionIcc.c
4       //   icc -O2 -std=c99 -mkl diffusionIcc.c
5       //   icc -O3 -std=c99 -mkl diffusionIcc.c
6       //   icc -O3 -std=c99 -mkl -qopt-report=1 -qopt-report-annotate=html diffusionIcc.c
7       // Execution:
8       //   ./a.out 'version' 'T' 'N'
9       //   ./a.out 1 1000 1024; ./a.out 2 1000 1024; ./a.out 3 1000 1024; ./a.out 4 1000 1024;
10
11      #include <stdio.h>
12      #include <stdlib.h>
13      #include <math.h>
14      #include <mkl.h>
15
16      void switchPointer(double** x1, double** x2);
17      void implementation1(int T, int N, double* C, double* Cnew);
18      void implementation2(int T, int N, double* C, double* Cnew);
19      void implementation3(int T, int N, double* C, double* Cnew);
20
21      int main(int argc, char* argv[])
22      {
```

```
INLINE REPORT: (main(int, char **))
  -> INLINE: (50,20) implementation1(int, int, double *, double *)
    -> INLINE: (81,5) switchPointer(double **, double **)
  -> INLINE: (51,20) implementation2(int, int, double *, double *)
    -> INLINE: (96,5) switchPointer(double **, double **)
  -> INLINE: (52,20) implementation3(int, int, double *, double *)
    -> INLINE: (113,5) switchPointer(double **, double **)

/Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(22,1):remark #34051: REGISTER ALLOCATION : [_main] /Users/modave/Google Drive/Teaching/

    Hardware registers
        Reserved     :    2[ rsp rip]
        Available    :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save  :    6[ rbx rbp r12-r15]
        Assigned     :   30[ rax rdx rcx rbx rsi rdi r8-r15 zmm0-zmm15]

    Routine temporaries
        Total         :     344
            Global    :     120
            Local     :     224
        Regenerable   :      31
        Spilled       :      22

    Routine stack
        Variables     :       4 bytes*
            Reads     :       2 [1.51e-01 ~ 0.2%]
            Writes    :       2 [1.51e-01 ~ 0.2%]
        Spills        :     136 bytes*
            Reads     :      40 [1.67e+00 ~ 1.7%]
            Writes    :      30 [1.43e+00 ~ 1.4%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

```
23
24        // Parameters
25        int version, T, N;
26        if(argc == 4){
27          version = atoi(argv[1]);
28          T = atoi(argv[2]);
29          N = atoi(argv[3]);
30        }
31        else{
32          printf("3 arguments nécessaires : version, T, N\n");
33          return 1;
34        }
35
36        // Initialization of arrays
37        double* C    = malloc(N*N*sizeof(double));
38        double* Cnew = malloc(N*N*sizeof(double));
39        for(int i=0; i<N; i++){
```

```
LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(39,3)
   remark #25460: No loop optimizations reported

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(40,5)
   <Peeled loop for vectorization>
   LOOP END

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(40,5)
      remark #15300: LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(40,5)
   <Remainder loop for vectorization>
   LOOP END
LOOP END
```

```
40          for(int j=0; j<N; j++){
41            double x = ((double)i)/(N-1.);
42            double y = ((double)j)/(N-1.);
43            C[i*N+j] = exp(-((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5))/0.01);
44          }
45        }
46
47        // Start CHRONO
48        const double timeBegin = dsecnd();
49
50        if(version == 1) implementation1(T, N, C, Cnew);
51        if(version == 2) implementation2(T, N, C, Cnew);
```

```
LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(90,3) inlined into /Users/modave/Google Drive/Teaching/SIM203/codes/codes
   remark #25460: No loop optimizations reported

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(91,5) inlined into /Users/modave/Google Drive/Teaching/SIM203/codes/co
      remark #25460: No loop optimizations reported

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(92,7) inlined into /Users/modave/Google Drive/Teaching/SIM203/codes
         remark #25439: unrolled with remainder by 4
      LOOP END

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(92,7) inlined into /Users/modave/Google Drive/Teaching/SIM203/codes
      <Remainder>
      LOOP END
   LOOP END
LOOP END
```

```
52         if(version == 3) implementation3(T, N, C, Cnew);
```

```
LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(107,3) inlined into /Users/modave/Google Drive/Teaching/SIM203/codes/code
   remark #25460: No loop optimizations reported

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(108,5) inlined into /Users/modave/Google Drive/Teaching/SIM203/codes/c
      remark #25460: No loop optimizations reported

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(109,7) inlined into /Users/modave/Google Drive/Teaching/SIM203/code
         remark #25439: unrolled with remainder by 4
      LOOP END

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(109,7) inlined into /Users/modave/Google Drive/Teaching/SIM203/code
      <Remainder>
      LOOP END
   LOOP END
LOOP END
```

```
53
54         // End CHRONO
55         const double timeEnd = dsecnd();
56         double timeTotal = timeEnd-timeBegin;
57         printf("%i %f %f\n", version, timeTotal, C[N*N/2 + N/2]);
58
59         return 0;
60       }
61
62       // Switch of pointers
63       void switchPointer(double** x1, double** x2){
```

```
INLINE REPORT: (switchPointer(double **, double **))

/Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(63,45):remark #34051: REGISTER ALLOCATION : [_switchPointer] /Users/modave/Google Drive

    Hardware registers
        Reserved     :    2[ rsp rip]
        Available    :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save  :    6[ rbx rbp r12-r15]
        Assigned     :    4[ rax rdx rsi rdi]

    Routine temporaries
        Total         :      12
            Global    :       0
            Local     :      12
        Regenerable   :       0
        Spilled       :       0

    Routine stack
        Variables     :       0 bytes*
            Reads     :       0 [0.00e+00 ~ 0.0%]
            Writes    :       0 [0.00e+00 ~ 0.0%]
        Spills        :       0 bytes*
            Reads     :       0 [0.00e+00 ~ 0.0%]
            Writes    :       0 [0.00e+00 ~ 0.0%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

```
64         double* tmp;
65         tmp = *x1;
66         *x1 = *x2;
67         *x2 = tmp;
68       }
69
70       // Implementation 1
71       void implementation1(int T, int N, double* C, double* Cnew){
```

```
INLINE REPORT: (implementation1(int, int, double *, double *))
  -> INLINE: (81,5) switchPointer(double **, double **)

/Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(71,60):remark #34051: REGISTER ALLOCATION : [_implementation1] /Users/modave/Google Dri

    Hardware registers
        Reserved     :    2[ rsp rip]
        Available    :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save  :    6[ rbx rbp r12-r15]
        Assigned     :   19[ rax rdx rcx rbx rbp rsi rdi r8-r15 zmm0-zmm3]

    Routine temporaries
        Total         :      53
            Global    :      30
            Local     :      23
        Regenerable   :       5
        Spilled       :       7

    Routine stack
        Variables     :       0 bytes*
            Reads     :       0 [0.00e+00 ~ 0.0%]
            Writes    :       0 [0.00e+00 ~ 0.0%]
        Spills        :       8 bytes*
            Reads     :       1 [1.75e-01 ~ 0.2%]
            Writes    :       1 [7.79e-02 ~ 0.1%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

```
72        double dx = 1./(double)(N-1);
73        double dt = 0.2*dx*dx;
74
75        for(int n=0; n<T; n++){
76          for(int j=1; j<(N-1); j++)
77            for(int i=1; i<(N-1); i++)
78              Cnew[N*i+j]
79                = (1 - 4*dt/(dx*dx)) * C[N*i+j]
80                + dt/(dx*dx) * ( C[N*(i+1)+j] + C[N*(i-1)+j] + C[N*i+(j+1)] + C[N*i+(j-1)] );
81          switchPointer(&C,&Cnew);
82        }
83      }
84
85      // Implementation 2
86      void implementation2(int T, int N, double* C, double* Cnew){
```

```
INLINE REPORT: (implementation2(int, int, double *, double *))
  -> INLINE: (96,5) switchPointer(double **, double **)

/Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(86,60):remark #34051: REGISTER ALLOCATION : [_implementation2] /Users/modave/Google Dri

    Hardware registers
        Reserved     :    2[ rsp rip]
        Available    :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save  :    6[ rbx rbp r12-r15]
        Assigned     :   25[ rax rdx rcx rbx rbp rsi rdi r8-r15 zmm0-zmm9]

    Routine temporaries
        Total        :      98
            Global   :      35
            Local    :      63
        Regenerable  :       5
        Spilled      :      10

    Routine stack
        Variables    :       0 bytes*
            Reads    :       0 [0.00e+00 ~ 0.0%]
            Writes   :       0 [0.00e+00 ~ 0.0%]
        Spills       :      56 bytes*
            Reads    :       7 [7.30e-01 ~ 0.7%]
            Writes   :       7 [7.30e-01 ~ 0.7%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

```
87        double dx = 1./(double)(N-1);
88        double dt = 0.2*dx*dx;
89
90        for(int n=0; n<T; n++){
```

```
LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(90,3)
   remark #25460: No loop optimizations reported

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(91,5)
      remark #25460: No loop optimizations reported

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(92,7)
         remark #25439: unrolled with remainder by 4
      LOOP END

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(92,7)
      <Remainder>
      LOOP END
   LOOP END
LOOP END
```

```
91          for(int i=1; i<(N-1); i++)
92            for(int j=1; j<(N-1); j++)
93              Cnew[N*i+j]
94                = (1 - 4*dt/(dx*dx)) * C[N*i+j]
95                + dt/(dx*dx) * ( C[N*(i+1)+j] + C[N*(i-1)+j] + C[N*i+(j+1)] + C[N*i+(j-1)] );
96          switchPointer(&C,&Cnew);
97        }
98      }
99
100     // Implementation 3
101     void implementation3(int T, int N, double* C, double* Cnew){
```

```
INLINE REPORT: (implementation3(int, int, double *, double *))
  -> INLINE: (113,5) switchPointer(double **, double **)

/Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(101,60):remark #34051: REGISTER ALLOCATION : [_implementation3] /Users/modave/Google Dr

    Hardware registers
        Reserved     :    2[ rsp rip]
        Available    :   39[ rax rdx rcx rbx rbp rsi rdi r8-r15 mm0-mm7 zmm0-zmm15]
        Callee-save  :    6[ rbx rbp r12-r15]
        Assigned     :   25[ rax rdx rcx rbx rbp rsi rdi r8-r15 zmm0-zmm9]

    Routine temporaries
        Total        :      97
            Global   :      32
            Local    :      65
        Regenerable  :       5
        Spilled      :      10

    Routine stack
        Variables    :       0 bytes*
            Reads    :       0 [0.00e+00 ~ 0.0%]
            Writes   :       0 [0.00e+00 ~ 0.0%]
        Spills       :      56 bytes*
            Reads    :       7 [7.30e-01 ~ 0.7%]
            Writes   :       7 [7.30e-01 ~ 0.7%]

    Notes

        *Non-overlapping variables and spills may share stack space,
         so the total stack size might be less than this.
```

```
102       double dx = 1./(double)(N-1);
103       double dt = 0.2*dx*dx;
104       double coef1 = dt/(dx*dx);
```

```
105       double coef2 = 1 - 4*coef1;
106
107       for(int n=0; n<T; n++){
```

```
LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(107,3)
   remark #25460: No loop optimizations reported

   LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(108,5)
      remark #25460: No loop optimizations reported

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(109,7)
         remark #25439: unrolled with remainder by 4
      LOOP END

      LOOP BEGIN at /Users/modave/Google Drive/Teaching/SIM203/codes/codes2sol/diffusionIcc.c(109,7)
      <Remainder>
      LOOP END
   LOOP END
LOOP END
```

```
108         for(int i=1; i<(N-1); i++)
109           for(int j=1; j<(N-1); j++)
110             Cnew[N*i+j]
111               = coef2 * C[N*i+j]
112               + coef1 * ( C[N*(i+1)+j] + C[N*(i-1)+j] + C[N*i+(j+1)] + C[N*i+(j-1)] );
113         switchPointer(&C,&Cnew);
114       }
115     }
```