1 – Optimisation C

00_base
Représentation matricielle d'une image.
Importance de la réflexion sur les valeurs possibles pour chaque variable qui détermine le type de stockage et donc la taille en mémoire ainsi que la complexité des opérations sur cette donnée.
L'utilisation de malloc est importante : les données seront stockées dans le tas et non pas dans le code, ce qui complique la tâche de l'unité d'exécution.
L'appel à malloc est non déterministe, on doit vérifier que la demande a été satisfaite et on doit indiquer que la ressource n'est plus utilisée à la fin du traitement.
Les valeurs sont ou non à 0, en fonction de la plateforme et du compilateur
La notation avec des crochets équivaut à un accès non prédictible pour l'unité d'exécution
La comparaison de deux valeurs avec < ou > est coûteuse : il faut faire une soustraction puis examiner le bit de signe (N) de l'Unité Arithmétique et Logique, cela correspond aussi à une approche non déterministe de l'algorithme ; par exemple si on sait qu'il faut 5 pas pour parcourir une distance l'approche classique correspond à se demander si on a fait assez de pas, l'approche déterministe calcul combien de pas sont à faire et en fait le nombre déterminé.

01_base_initialisee
Approche de l'algorithmique avancée en répondant aux questions : Combien de fois / point de départ / action / passage à la prochaine itération
L'utilisation d'un décompteur est plus économique : on teste le bit Z (Zero) de l'Unité Arithmétique et Logique qui a été mis à jour lors de la dernière décrémentation qui est plus rapide qu'une comparaison.

01bis → 04 : mise en pratique
04 (ligne oblique) : attention à ce que la comparaison de WIDTH avec HEIGHT soit hors de la boucle car cette condition n'évolue pas au fur et à mesure du traitement de données ; c'est un invariant de boucle.

06 BMP read write :
Introduction de l'utilisation de projet : fichier qui contient la configuration de l'éditeur, les liens vers l'ensemble des fichiers nécessaire et les règles données au compilateur et à l'éditeur de liens pour produire l'exécutable.
Utilisation de code qui permet de lire et sauvegarder des images au format BMP non compressé.
On peut tracer des lignes comme précédemment, l'image sauvegardée porte ces modifications.

TD1 : Mise en pratique avec miroir d'axe horizontal, puis vertical : commencer par compléter le tableau à quatre colonnes puis coder.

TD2 : Transformation en négatif de l'image source. Premier invariant algorithmique : la transformation des données de l'image ne dépend pas de l'endroit où se situe la donnée, donc de l'évolution du contexte du traitement.
On va transcrire l'opération dans une Lookup Table (LUT ou abaque) qui ne sera calculée qu'une seule fois. On peut même se permettre d'utiliser du code peu optimisé mais facilement lisible et modifiable : ici il n'y a que 256 itérations à faire pour traiter n'importe quelle quantité de données.
On remplace donc du code par de la mémoire, cette démarche ne vaut que si le temps passé à faire le traitement est bien supérieur à celui nécessaire pour construire la LUT.

TD3 : Correction lumière et contraste
Même principe que TD2 au niveau de la LUT, cette fois pour un code bien plus coûteux à exécuter du fait des tests, calculs sur des nombres décimaux et des changements de types.

07 Mesure du temps
Le système de mesure du temps d'exécution :
Attention à ne pas mesurer des temps trop faibles qui peuvent être du même ordre que celui passé à exécuter d'autres processus. Pour ce faire, jouer sur deux paramètres : la taille des données à traiter à chaque itération et le nombre d'itérations.
Dans cet environnement le main est figé. Il initialise notre contexte d'exécution (structure process_data_t) avec les paramètres de la ligne de commande en appelant init_process_data. C'est le moment pour nous d'initialiser tout ce qui est constant lors du traitement. Puis intervient la mesure du temps d'exécution d'une fonction de notre cru exécutée LOOPS fois. Un pointeur de fonction permet de modifier la fonction appelée lors de la mesure, cela permet d'expérimenter plus facilement.

Le traitement mesuré :
Le cahier des charges de ce code est de générer une image telle que, pour chaque ligne :
- les valeurs des pixels des trois premiers pixels sont le numéro de ligne, numéro de ligne + 1 et numéro de ligne + 2
- ensuite 256 pixels avec un dégradé qui part de 0 (noir) jusqu'à 255 (blanc)
- le reste de la ligne vaut 255 pour la première ligne, 254 pour le deuxième, ainsi de suite jusqu'à 0 puis repasse à 255 et boucle sur toute la hauteur de l'image générée.
basic_data_processing (28s) : méthode académique, claire mais dont l'approche non déterministe pénalise le temps d'exécution.
basic_data_processing_v2 (22s) : approche plus déterministe - d'où le gain de temps - on rempli les colonnes qui sont homogènes entre elles.
faster_data_processing (1,2s) : approche déterministe combinée à un accès de la mémoire dans le sens qui convient au fonctionnement de la cache mémoire du processeur. L'abandon de la notation avec crochets rend l'exécution plus facilement transformable par l'unité d'exécution de façon à l'adapter à ce qui sera le plus efficient.
faster_data_processing_gradient_copy (0,8s) : on note que le gradient horizontal est un invariant algorithmique, aussi il est possible de le pré-calculer et de simplement le copier dans la zone appropriée.

08 Lumière contraste
Mesure des optimisations liées à l'utilisation de la LUT
basic_data_processing (63,7s) : méthode académique
pointer_data_processing (62,97s) : soigner l'accès à la mémoire ne change que très peu de choses. Le goulot d'étranglement est donc ailleurs : le test ou les changements de type entre double et uint8_t
pointer_data_processing_lut (16,7s) : L'utilisation de la LUT permet d'accélérer vraiment l'exécution car l'invariant algorithmique a été pré-calculé. L'utilisation d'une variable locale pour accéder à la LUT n'a que peu d'effet.

09
On imagine un traitement fictif sur 8 bits non signés qui ajoute 127 aux valeurs inférieures ou égales à 128 et divise par deux les autres valeurs.
basic_data_processing (46,78s) : méthode académique

pointer_data_processing (38,52s) : permet de constater le gain de l'accès mémoire lisible par l'unité d'exécution. On déduit que dans le cas de la correction lumière contraste c'est le transtypage qui était le goulot d'étranglement.
pointer_reverse_data_processing (37,29s) : parcourir la mémoire à l'envers n'a ici que peu d'impact, on peut imaginer que la gestion de la cache se fait en lisant à des adresses multiples d'un certaine valeur.
pointer_random_access_data_processing (48,34s) : nous masquons ici l'ordre dans lequel accéder à la mémoire ce qui pénalise énormément l'efficacité.
lut_array_data_processing (27,73s) : conserve la notation crochet pour accéder aux données mais utilise une LUT ce qui donne un meilleur gain que de soigner l'accès à la mémoire. Comme quoi l'exécution du code reste un frein à l'efficacité, accéder à de la mémoire reste plus rapide si le traitement peut s'y préter.
lut_array_data_processing_v2 (27,72s) : idem précédent mais avec l'utilisation de variables locales ce qui ne change pas les choses
lut_pointer_data_processing (30,6s, avec locale : 24,07s) : ici l'utilisation d'une variable locale change beaucoup la donne. Sans elle, c'est moins rapide que la version avec notation crochets.

2 – Langages objets, mémoire managée

Méthodes classiques :
- méthodes utilitaires à mettre en méthodes de classe pour économiser le contexte
- invariants de boucle
- factorisation dans les calculs

Gestion de la mémoire
- réutiliser les instances, quitte à rajouter une méthode d'initialisation des instances pour le recyclage
- StringBuilder vs String ; instance immuable vs muable
- La gestion de la mémoire automatique :
→ allocations rapides
→ collectes lentes voire bloquantes (gel de l'interface ou des sockets, etc).
Pour faciliter la vie du garbage collector : avoir des instances temporaires vraiment temporaires et ne modifier l'état des instances plus persistantes que si on est absolument sûr de la nécessité (i.e. il vaut mieux mettre des tests plutôt que de faire des modifications inutiles) ou bien cloner pour modifier la version plus récente ; un if est moins cher qu'une racine supplémentaire à explorer lors de la collecte.
- la gestion de la mémoire managée n'affranchit pas de la libération explicite des ressources externes.

Design :
- Bonne utilisation de l'héritage et du polymorphisme : exemple 2 (Forme) puis 3 (GUI)
- Bonne utilisation de l'encapsulation pour éviter de se poser constamment des questions sur la validité de l'état d'une instance (checkValidity)
- Aller encore plus loin avec la configuration du programme qui se traduit dans la nature concrète d'instances à qui on délègue le travail effectif : exemple du logiciel de dessin et du traitement de données.