Aritmética de precisão arbitrária para OpenCL

Renan Oliveira Moreira

Cronograma

- Introdução sobre:
  - Aritmética de precisão arbitrária
  - OpenCL
- Adição
  - Algoritmo
  - Adição em OpenCL
  - Adição em Java

Cronograma

- Multiplicação
  - Algoritmo
  - Multiplicações em OpenCL
  - Multiplicação em Java
- Resultados
- Conclusão

Precisão arbitrária

- Erros de arredondamento
- Precisão dos números é variável
- Limitada pela memória da máquina
- Demorada

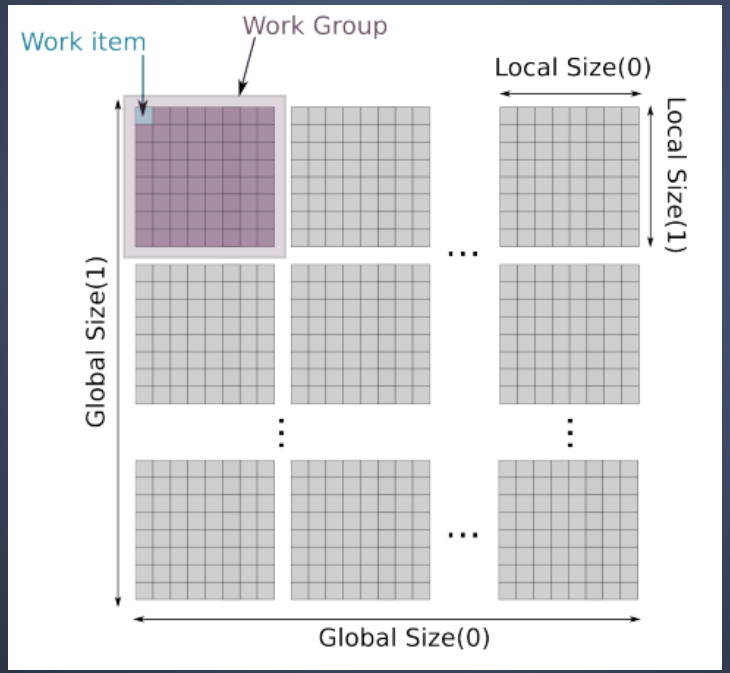
OpenCL

- Computação em paralelo
- GPGPU
- OpenCL C
- *Host* e *Kernel*

OpenCL

- *API* JOCL
- *Compute units* e *Processing Element*
- Dimensões em OpenCL
- *Work-Items* e *Work-Groups*

OpenCL

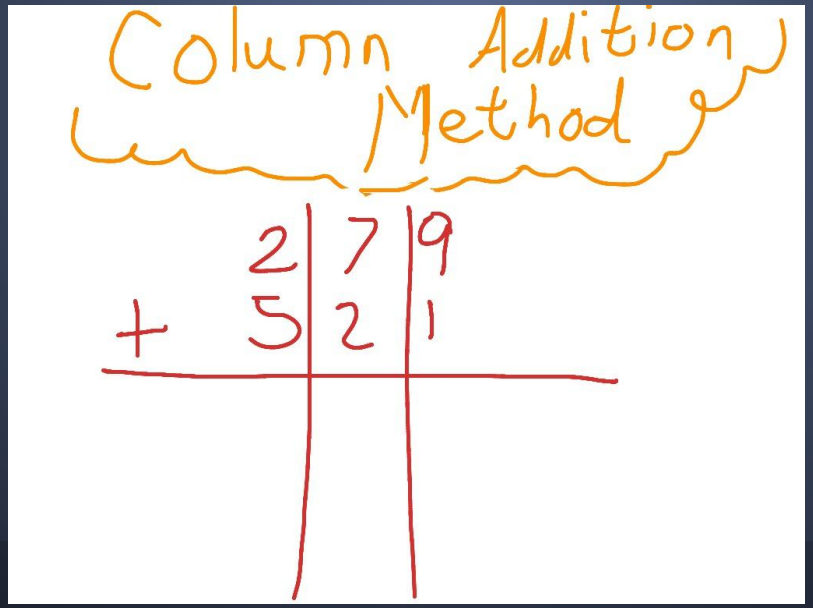

Work item
Work Group
Local Size(0)
Local Size(1)
Global Size(1)
Global Size(0)

OpenCL

```
if (operacao == "somaParaleloKernel"){
    global_work_size = new long[]{numeroDeWork};
    local_work_size = new long[]{1};
}else if (operacao == "multiplicacaoParaleloKernelDec2D"){
    global_work_size = new long[]{tamanhoNumero1,tamanhoNumero2};
    local_work_size = new long[]{1,1};
}else if (operacao == "multiplicacaoParaleloKernelDec1D"){
    global_work_size = new long[]{tamanhoNumero1};
    local_work_size = new long[]{1};
}else if (operacao == "multiplicacaoParaleloKernelDec0D"){
    global_work_size = new long[]{1};
    local_work_size = new long[]{1};
}
```

Algoritmo de Adição

- Método da adição das colunas



Adição em OpenCL

- Número será separado em colunas
- Colunas terão tamanho 18
- Número de *work-items* é igual a quantidade de colunas do maior número

Adição em OpenCL

- Global_id representa cada coluna

```
// Soma
private static String codigoOpenCLSoma =
    "__kernel void "+
    "somaParaleloKernel(__global const long *numero1,"+
    "                   __global const long *numero2,"+
    "                   __global long *saida)"+
    "{"+
    "    int gid = get_global_id(0);"+
    "    saida[gid] = numero1[gid] + numero2[gid];"+
    "}";
```

Adição em OpenCL

- Passagem de carry
- Preenchimento com zero
- Concatenação

Adição em Java

- Parecido com a realizada em OpenCL
- Utilização de um *loop* “for” ao invés de global_id
- Passagem de carry, preenchimento com zero e concatenação

Algoritmo de Multiplicação

- Método tradicional x métodos mais eficientes
- Multiplicar colunas
- Posição “n”
- Passagem de carry para “n+1”

Multiplicação em OpenCL

- Colunas terão tamanho 9
- Multiplicação com 2 *loops*
- 1 Dimensão com 1 *loop*
- 2 Dimensões com nenhum *loop*

Multiplicação em OpenCL

- Multiplicação com 2 *loops*

```
140    private static String codigoOpenCLMultiplicacaoDecimal0D =
141            "__kernel void "+
142            "multiplicacaoParaleloKernelDec0D(__global const long *numero1,"+
143            "       __global const long *numero2,"+
144            "       __const int tamanhoNumero1,"+
145            "       __const int tamanhoNumero2,"+
146            "       __global long* saida)" +
147            "{" +
148            " for (int i = 0; i < tamanhoNumero1 + tamanhoNumero2; i++)" +
149            "       saida[i] = 0;" +
150            " long aux,aux2,aux3; " +
151            " for (int a = 0; a < tamanhoNumero1 ; a++){"+
152            "   for (int b = 0; b < tamanhoNumero2  ; b++){" +
153            "       aux = numero1[a] * numero2[b];" +
154            "       aux2 = (aux % 1000000000);" + // mod, remainder
155            "       aux3 = aux / 1000000000;" +
156            "       saida[a+b] = saida[a+b] + aux2 ; " +
157            "       saida[a+b+1] = saida[a+b+1] + aux3; " +
158            "       if (saida[a+b] >= 1000000000){" +
159            "           aux2 = saida[a+b] % 1000000000;"    +
160            "           aux3 = saida[a+b] / 1000000000;" +
161            "           saida[a+b] = aux2;" +
162            "           saida[a+b+1] = saida[a+b+1] +aux3;" +
163            "       }"  +
164
165            "   }"+
166            " }" +
167            "}";
```

Multiplicação em OpenCL

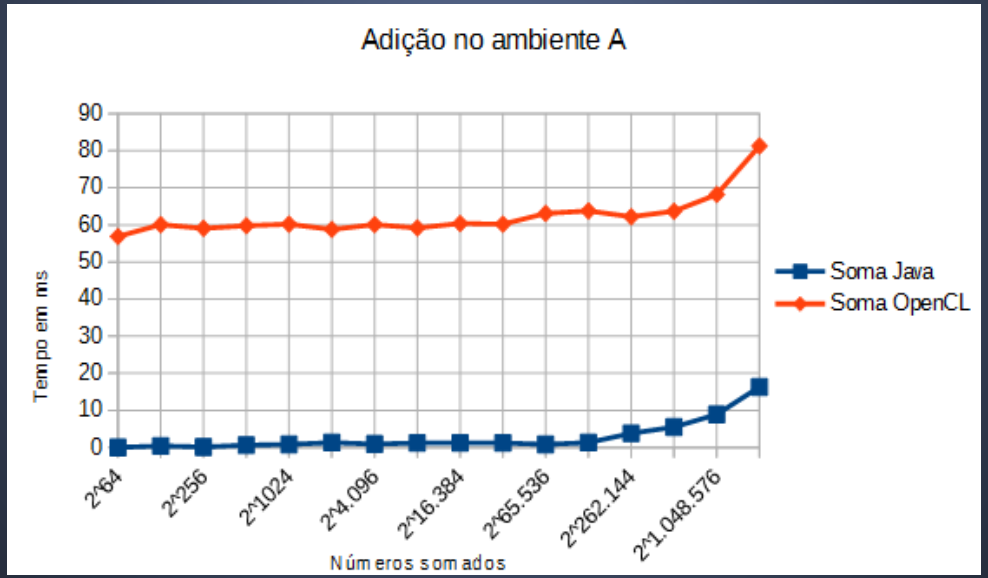
- Multiplicação com 1 *loop*
- Funções atómicas e semáforos
- *Work-items* não são sincronizados

Multiplicação em Java

- Idêntica à utilizada em OpenCL com 2 *loops*

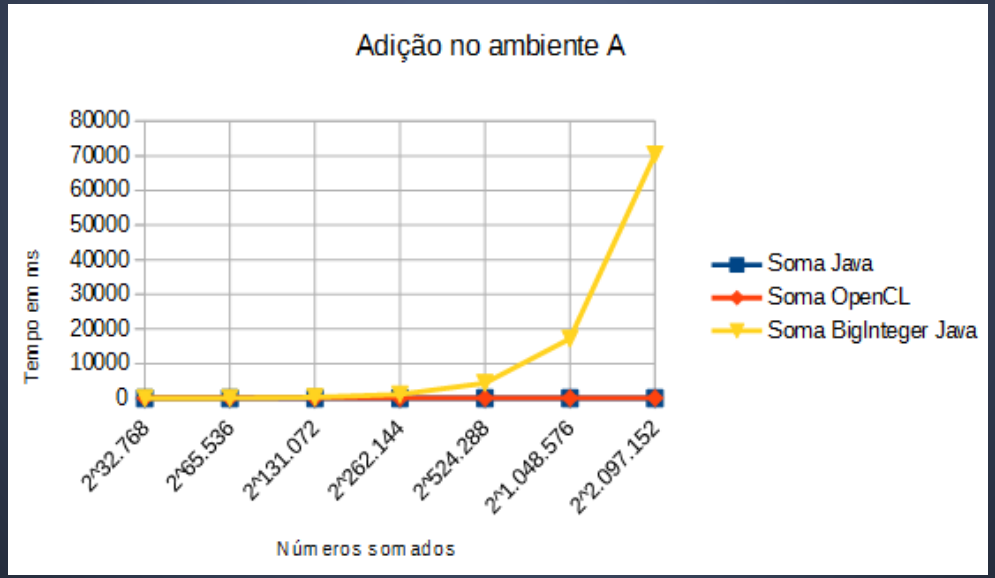
Resultados

- Adição



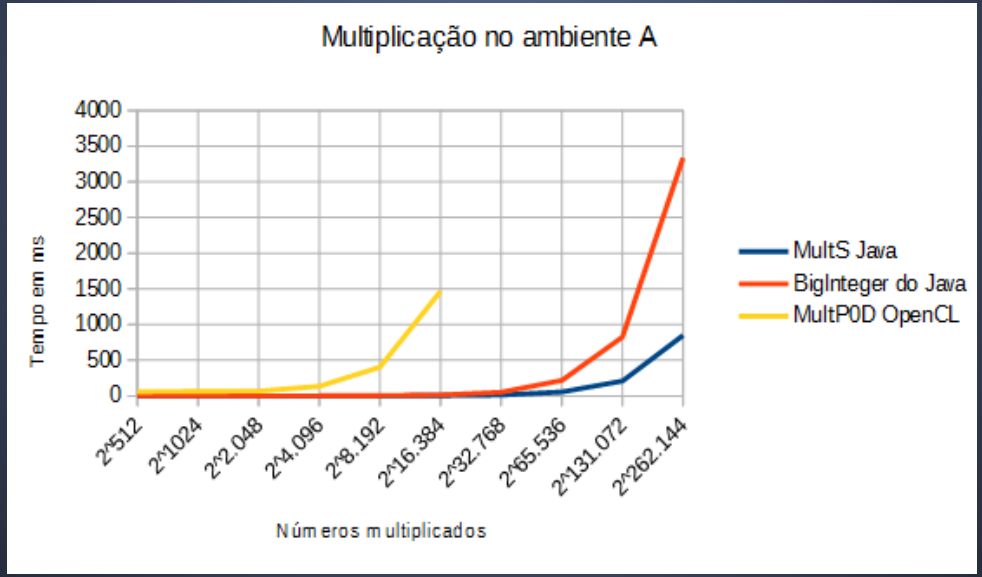
Resultados

- Adição



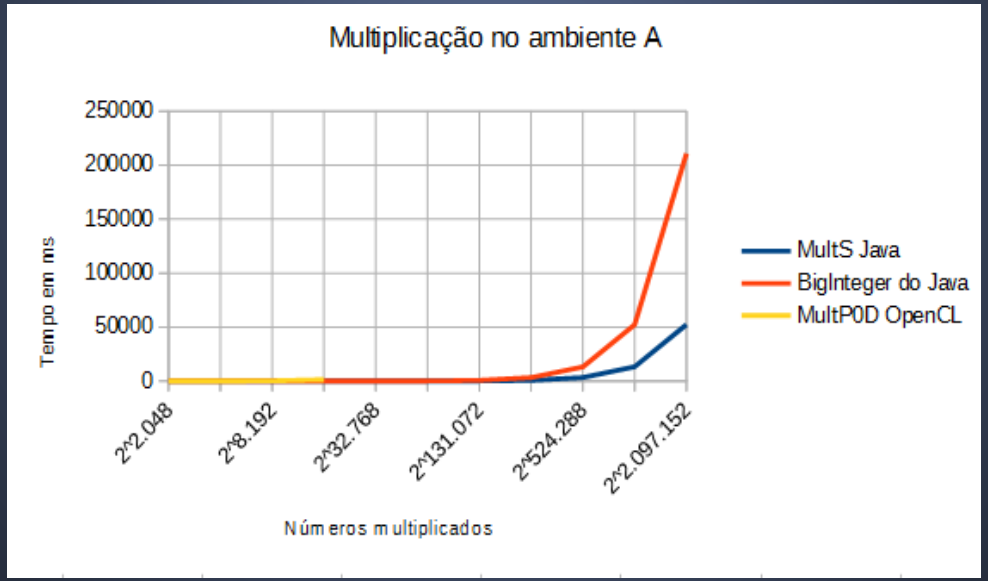
Resultados

- Multiplicação



Resultados

- Multiplicação



Conclusão

- Operações de adição apresentaram desempenhos semelhantes
- Multiplicação em Java mais rápido
- OpenCL apresentou falta de recursos

Conclusão

- Não utilização do algoritmo de Karatsuba
- Não utilização de *work-groups*
- Algoritmo de multiplicação em OpenCL não utilizou paralelização