Tips – VulkanCompute XOP

この XOP は、Vulkan を使って、GLSL シェーダーを GPU 上で実行できる実験的な XOP です。
この XOP では、GPU バッファとして　Igor Pro のウェーブを、プッシュ定数として Igor Pro のスカラーを使います。
要素ごとの計算負荷が高く、高度に並列化された処理を GPU にオフロードしたい場合に便利です。

VulkanCompute [/IN=*inList* /OUT=*outList* /PC=*pcWave* /SPV=*spvWave* /GRP={gx, *gy*, *gz*} /ENT=*entryName* /DBL /Q /Z] [*shaderSource*]

VulkanCompute コマンドは、Vulkan を使って、ユーザーが指定した GLSL コンピュートシェーダーを GPU 上で実行します。
ウェーブは GPU ストレージバッファ（SSBO）としてバインドされ、スカラー値はプッシュ定数として渡されます。
シェーダーは、GLSL ソーステキスト（shaderSource）として、またはプリコンパイル済みの SPIR-V（/SPV）として指定されます。
後者の場合、GLSL コンパイラは呼び出されません。

VulkanCompute は、VulkanCompute XOP によって追加される独立したコマンドです。
これを使うには、Vulkan 対応の GPU とドライバーが必要です。
シェーダーが従わなければならないバッファバインディングの規則については、「詳細」のセクションを参照してください。

パラメーター

shaderSource は、文字列形式の GLSL コンピュートシェーダーです。
/SPV オプションでプリコンパイル済みの SPIR-V が指定されていない限り、これは必須です。
両方が指定された場合、/SPV オプションが優先され、*shaderSource* は無視されます。

inList と *outList* で指定されたウェーブは、現在のデータフォルダー内で解決されます。

フラグ

/IN=*inList*　セミコロンで区切られた入力ウェーブの名前の一覧です。これらは読み取り専用ストレージバッファとしてバインドされます。ウェーブは実数型のウェーブである必要があり、現在のデータフォルダー内に存在している必要があります。

/OUT=*outList*　出力ウェーブの名前をセミコロンで区切ったリストです。これらは読み書き可能なストレージバッファとしてバインドされます。出力ウェーブは結果を受け取り、「変更済み」とマークされます。

/PC=*pcWave*　オプションのウェーブです。その値は、シェーダーの push_constant ブロックとなります。値は 32 ビット浮動小数点数に変換され、オフセット 0 から連続して配置されます。最大 32 個の値（128 バイト）が使われます。

/SPV=*spvWave*　オプションのプリコンパイル済み SPIR-V モジュールです。符号なし 32 ビット整数のウェーブ形式で提供されます（例えば <VulkanCompileShader> によって生成されたもの）。これが指定されている場合、GLSL コンパイラはスキップされ、*shaderSource* は不要となります。モジュールは使用前に検証されます。

/GRP={*gx*, *gy*, *gz*} 各次元で割り当てるワークグループの数です。省略した場合、*gx* = ceil(*maxElements* / 64)、*gy* = *gz* = 1 となり、layout(local_size_x = 64) で宣言されたシェーダーに対応します。

/ENT=*entryName* シェーダーのエントリポイント関数の名前です。デフォルトは main です。

/DBL 64 ビット浮動小数点（fp64）バッファを要求します。これは、ソースウェーブが倍精度であり、かつデバイスが shaderFloat64 機能をサポートしている場合にのみ有効となります。それ以外の場合はエラーが返されます。

/Q 静寂モードです。シェーダーのコンパイルログや SPIR-V 検証ログが履歴に出力されるのを抑制します。

/Z エラーが発生しても処理は中断されません。その代わりに、ステータスコードは出力変数 V_flag に返されます。

詳細

シェーダーは、以下のバインディング規約に従わなければなりません。

- すべてのバッファはディスクリプター set 0 にあります。
- 0 から N-1 までのバインディングは、記載された順序で /IN ウェーブとなります。
- N から N+M-1 までのバインディングは、記載された順序で /OUT ウェーブとなります。
- 各バッファは、std430 ストレージバッファとして宣言されています。

ウェーブ要素の順序は、Igor Pro の内部保存形式と一致する「列優先線形」（行、次に列、次にレイヤー、次にチャンク）です。
gl_GlobalInvocationID に基づいて、バッファへのインデックス付けは自身で行ってください。

デフォルトでは、バッファは 32 ビット浮動小数点型を使います。
fp64 を使うには、/DBL オプションを指定してください（/DBL オプションを参照）。

シェーダーが単精度で多くの値を蓄積する場合は注意が必要です。
32 ビット浮動小数点数は、有効桁数が約 7 桁しかないため、累積和、内積、または長い反復ループでは、加算される項に比べて累積値が大きくなるにつれて精度が失われ、小さな丸め誤差が累積する可能性があります。
これは、カオス的またはフィードバック的な再帰処理ではさらに深刻になります。
そこでは、fp32 のわずかな差が増幅され、結果が倍精度（CPU）の参照値から著しく逸脱する可能性があります。
誤差を低減するには、条件の良い式（例えば、無制限の累積和ではなく、同じ絶対値を持つ項の平均など）を優先するか、中間値の絶対値が同等になるように累積を行うか、デバイスが shaderFloat64 をサポートしていて、速度よりも精度が重要である場合は /DBL を使ってください。

プッシュ定数（/PC）は常に float 型に変換されます。
一般的な慣習として、要素数を最初のプッシュ定数に格納し、シェーダー内でそれを float 型として読み取り、境界検査のために呼び出しインデックスをキャストします。
以下の例に示す通りです。

戻り値として、*V_flag* は成功した場合は 0 に、XOP エラーが発生した場合は 0 以外のエラーコードに設定されます。
/Z オプションを指定した場合、操作は中断されないため、呼び出し元は *V_flag* を確認することができます。

この処理では、呼び出しのたびにランタイムシェーダーコンパイラを使って GLSL 文字列をコンパイルします。
呼び出しごとのこのオーバーヘッドを回避するには、<VulkanCompileShader> を使って一度プリコンパイルし、その結果として生成された SPIR-V ウェーブを /SPV オプションで渡してください。

GPU オフロードが有効なのは、演算密度が高い（要素あたりの演算数が多い）シェーダーの場合、あるいは同じデータが多数のカーネルによって処理される場合に限られます。
計算量は少ないものの大量のデータを移動させる単純な要素ごとの式は、GPU との間でデータをやり取りするコストがかかるため、通常は CPU 上で実行した方が高速になります（例えば、MatrixOp や FastOp を使う場合など）。

例

【例１】
ウェーブにスカラーを乗算します。入力１つ、出力１つ、プッシュ定数１つです。これが最も単純な完全なシェーダーです。

```
Function DemoGPUScale()
    Make/O/N=1024/S src = p, dst

    // プッシュ定数：[0] = スケール、[1] = ポイント数（いずれも浮動小数点数）
    Make/O/N=2/S pc = {2.5, 1024}

    String sh = ""
    sh += "#version 450¥n"
    sh += "layout(local_size_x = 64) in;¥n"
    sh += "layout(std430, set=0, binding=0) readonly  buffer In0  { float a[]; };¥n"
    sh += "layout(std430, set=0, binding=1) writeonly buffer Out0 { float r[]; };¥n"
    sh += "layout(push_constant) uniform PC { float scale; float n; } pc;¥n"
    sh += "void main() {¥n"
    sh += "¥tuint i = gl_GlobalInvocationID.x;¥n"
    sh += "¥tif (float(i) < pc.n) {¥n"
    sh += "¥t¥tr[i] = a[i] * pc.scale;¥n"
    sh += "¥t}¥n"
    sh += "}¥n"

    VulkanCompute /IN="src" /OUT="dst" /PC=pc /Z sh
    if (V_flag != 0)
        Print "VulkanCompute failed, V_flag =", V_flag
        return -1
    endif
    // dst は src * 2.5 に等しくなる
End
```

【例２】
２つのウェーブを合成します：waveC = varA * waveA * waveB + varB。入力が２つ、出力が１つであるため、出力はバインディング２に割り当てられます。

```
Function DemoGPUCombine()
    Variable varA = 3, varB = 5
    Make/O/N=1000/S waveA = p, waveB = 2*p, waveC

    // プッシュ定数：[0] = n, [1] = varA, [2] = varB
    Make/O/N=3/S pc
    pc[0] = numpnts(waveA); pc[1] = varA; pc[2] = varB

    String sh = ""
    sh += "#version 450¥n"
    sh += "layout(local_size_x = 64) in;¥n"
    sh += "layout(std430, set=0, binding=0) readonly  buffer InA  { float a[]; };¥n"
    sh += "layout(std430, set=0, binding=1) readonly  buffer InB  { float b[]; };¥n"
    sh += "layout(std430, set=0, binding=2) writeonly buffer OutC { float c[]; };¥n"
    sh += "layout(push_constant) uniform PC { float n; float varA; float varB; } pc;¥n"
    sh += "void main() {¥n"
    sh += "¥tuint i = gl_GlobalInvocationID.x;¥n"
    sh += "¥tif (float(i) < pc.n) {¥n"
    sh += "¥t¥tc[i] = pc.varA * a[i] * b[i] + pc.varB;¥n"
```

```
    sh += "¥t}¥n"
    sh += "}¥n"

    VulkanCompute /IN="waveA;waveB" /OUT="waveC" /PC=pc /Z sh
    Print "V_flag =", V_flag
End
```

【例３】

演算中心のカーネル（GPU が真価を発揮する場面）です。各要素で超越関数演算の内部ループが実行されるため、このシェーダーではデータ転送よりも演算が主となります。

```
Function DemoGPUHeavy()
    Variable iters = 2000
    Make/O/N=200000/S heavyIn = sin(p*0.001) + 0.5, heavyOut

    // プッシュ定数： [0] = n, [1] = 要素ごとの反復回数
    Make/O/N=2/S pc
    pc[0] = numpnts(heavyIn); pc[1] = iters

    String sh = ""
    sh += "#version 450¥n"
    sh += "layout(local_size_x = 64) in;¥n"
    sh += "layout(std430, set=0, binding=0) readonly  buffer InA  { float a[]; };¥n"
    sh += "layout(std430, set=0, binding=1) writeonly buffer OutC { float c[]; };¥n"
    sh += "layout(push_constant) uniform PC { float n; float iters; } pc;¥n"
    sh += "void main() {¥n"
    sh += "¥tuint i = gl_GlobalInvocationID.x;¥n"
    sh += "¥tif (float(i) < pc.n) {¥n"
    sh += "¥t¥tfloat x = a[i];¥n"
    sh += "¥t¥tfloat acc = 0.0;¥n"
    sh += "¥t¥tint m = int(pc.iters);¥n"
    sh += "¥t¥tfor (int k = 1; k <= m; k++) {¥n"
    sh += "¥t¥t¥tfloat fk = float(k);¥n"
    sh += "¥t¥t¥tacc += sin(x*fk)*cos(x/fk) + sqrt(abs(x)+fk)*exp(-x*x*0.001);¥n"
    sh += "¥t¥t¥tx = x*0.9999 + 0.0001*acc;¥n"
    sh += "¥t¥t}¥n"
    sh += "¥t¥tc[i] = acc;¥n"
    sh += "¥t}¥n"
    sh += "}¥n"

    VulkanCompute /IN="heavyIn" /OUT="heavyOut" /PC=pc /Z sh
    Print "V_flag =", V_flag
End
```

【例４】

シェーダーをプリコンパイルして再利用します。<VulkanCompileShader> を使って GLSL を一度コンパイルし、その後 /SPV オプションを指定して何度も実行します。これにより、呼び出しごとのコンパイルが省略されます。

```
Function DemoGPUCached()
    Variable varA = 3, varB = 5, i, reps = 200
    Make/O/N=100000/S waveA = p, waveB = 2*p, waveC
    Make/O/N=3/S pc
    pc[0] = numpnts(waveA); pc[1] = varA; pc[2] = varB

    // "spv" という名前の符号なし 32 ビット SPIR-V ウェーブとして１回コンパイルする
    String sh = ""
    sh += "#version 450¥n"
    sh += "layout(local_size_x = 64) in;¥n"
    sh += "layout(std430, set=0, binding=0) readonly  buffer InA  { float a[]; };¥n"
    sh += "layout(std430, set=0, binding=1) readonly  buffer InB  { float b[]; };¥n"
    sh += "layout(std430, set=0, binding=2) writeonly buffer OutC { float c[]; };¥n"
    sh += "layout(push_constant) uniform PC { float n; float varA; float varB; } pc;¥n"
```

```
    sh += "void main() {¥n"
    sh += "¥tuint i = gl_GlobalInvocationID.x;¥n"
    sh += "¥tif (float(i) < pc.n) { c[i] = pc.varA * a[i] * b[i] + pc.varB; }¥n"
    sh += "}¥n"

    VulkanCompileShader /DEST="spv" /Z sh
    if (V_flag != 0)
        Print "compile failed, V_flag =", V_flag
        return -1
    endif
    WAVE/I spv

    // コンパイル済みの SPIR-V を複数のディスパッチで再利用する（再コンパイルは行わない）
    for (i = 0; i < reps; i += 1)
        VulkanCompute /IN="waveA;waveB" /OUT="waveC" /PC=pc /SPV=spv /Z
    endfor
    Print "done, V_flag =", V_flag
End
```

参照

ウェーブに関する CPU ベースの計算用の MatrixOp、FastOp、MultiThread

VulkanCompileShader [/ENT=*entryName* /DEST=*destName* /Q /Z] *shaderSource*

VulkanCompileShader コマンドは、GLSL コンピュートシェーダーを SPIR-V にコンパイルし、その結果を符号なし 32 ビット整数のウェーブに格納します。
このウェーブは、/SPV フラグを指定して <VulkanCompute> に渡すことができ、これにより、呼び出しのたびにシェーダーが再コンパイルされるのを回避できます。

コンパイルされた SPIR-V はデバイスやドライバに依存しないため、出力ウェーブを保存して、別のセッションや他のマシンでも再利用することができます。

パラメーター

shaderSource は、文字列形式の GLSL コンピュートシェーダーです。
これは必須です。

フラグ

/ENT=*entryName*　コンパイルするシェーダーのエントリポイント関数の名前です。デフォルトは main です。

/DEST=*destName*　現在のデータフォルダー内に作成する出力ウェーブの名前です。デフォルトは W_SPIRV です。その名前で既に存在するウェーブがある場合は上書きされます。

/Q　静寂モードです。シェーダーのコンパイルまたは検証ログが履歴に出力されるのを抑制します。

/Z　エラーが発生しても処理は中断されません。その代わりに、ステータスコードは出力変数 V_flag に返されます。

詳細

宛先ウェーブは、SPIR-V ワードをデータポイントとする符号なし 32 ビット整数ウェーブとして作成されます。
その長さは、コンパイラによって生成された SPIR-V ワードの数です。

コンパイルされたモジュールは、検証が行われます（ビルドで検証機能が利用可能な場合）。
コンパイルまたは検証に失敗したシェーダーはエラーを発生させ、出力ウェーブは生成されません。

戻り値として、V_flag は成功した場合は 0 に、失敗した場合は 0 以外の XOP エラーコードに設定されます。
/Z オプションを指定した場合、操作は中断されないため、呼び出し元は V_flag を確認することができます。

この結果を使うには、宛先のウェーブに対するウェーブ参照を宣言し、それを /SPV オプションとともに <VulkanCompute> に渡します。
シェーダーは SPIR-V にベイクされているため、VulkanCompute の呼び出し時に shaderSource 文字列を指定する必要はありません。

例

シェーダーを一度コンパイルして、それをディスパッチします（<VulkanCompute> の【例４】も参照してください）。

```
Function DemoCompileAndRun()
    Make/O/N=1024/S src = p, dst
    Make/O/N=2/S pc = {2.5, 1024}

    String sh = ""
    sh += "#version 450¥n"
    sh += "layout(local_size_x = 64) in;¥n"
    sh += "layout(std430, set=0, binding=0) readonly  buffer In0  { float a[]; };¥n"
    sh += "layout(std430, set=0, binding=1) writeonly buffer Out0 { float r[]; };¥n"
    sh += "layout(push_constant) uniform PC { float scale; float n; } pc;¥n"
    sh += "void main() {¥n"
    sh += "¥tuint i = gl_GlobalInvocationID.x;¥n"
    sh += "¥tif (float(i) < pc.n) { r[i] = a[i] * pc.scale; }¥n"
    sh += "}¥n"

    // SPIR-V ウェーブ "scaleSPV" にコンパイルする
    VulkanCompileShader /DEST="scaleSPV" /Z sh
    if (V_flag != 0)
        Print "compile failed, V_flag =", V_flag
        return -1
    endif
    WAVE/I scaleSPV

    // プリコンパイル済みモジュールを使ってディスパッチする
    VulkanCompute /IN="src" /OUT="dst" /PC=pc /SPV=scaleSPV /Z
    Print "V_flag =", V_flag
End
```